People Are Our Most Important Product

TR86-015

July, 1986

Frederick P. Brooks, Jr.

The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall, 083A
Chapel Hill, NC 27599-3175

People Are Our Most Important Product

Frederick P. Brooks, Jr.
University of North Carolina at Chapel Hill

1. Introduction

The function of a keynote speech (if any) should be to give perspective. Coming from outside the software engineering research field, but from within the computer field, I would like to offer an outsider's perspective on some current software engineering curricula proposals.

Let me start with a disclaimer. Since writing *The Mythical Man-Month*, I have not worked in software engineering management nor in software engineering research. Everyone here is more current in the field than I. I am a lifetime fan of computers and of software engineering; I teach a course in the subject, and I try to stay up-to-date in the field. But I am not really working in it.

The peak year in sales for *The Mythical Man-Month* was only two years ago. Yet the book was written in 1975, about an experience in 1963-65. The fact that it has the slightest relevance now is a sad comment on the progress of the discipline.

Wave After Wave

In the some 40 years since I first became interested in computers, we have seen seven revolutions, the first of which is the computer revolution represented by the Harvard Mark I. I was 13 when the Mark I was introduced, and watching with big eyes. That was the first I had ever heard of the idea of a computer. I decided that it was the exciting thing, and I started heading that way, [Figure 1].

Second, came electronic computers and the invention of assemblers and interpreters. In 1952, I had a chance to learn to program (in octal absolute) on the not-quite-delivered, vacuum-tube-based IBM 701. That experience was a major milestone for me.

The third revolution was brought by the transistor and Fortran – for me, that meant three years helping design Stretch. The System/360 was another major milestone for me. It represented the fourth revolution – integrated circuits and mandatory operating systems.

The fifth revolution brought minicomputers and the concurrent development of communications as an inherent part of most computer systems. The most recent revolution involves microcomputers and – one of the most important factors today – the mass marketability of microcomputer software and its corollary, packaged application programs.

Revolutions

Hardware	Software
1. Computers	
2. Electronics	Assemblers
3. Transistors	Compilers
4. Integrated Circuits	Operating Systems
5. Minicomputers	Communications
6. Microprocessors	Programming Environments
7. Mass-Market PC's	Packaged Applications

Figure 1

Twenty Tries at a Software Project Course

One of the things I did as soon as I got to Chapel Hill, was to start the kind of one-semester, small-team, classical project course that John Bentley,[1] says in his paper is not the right way to do it. I think everyone agrees it would be better as a two-semester project course.

Except for two years on sabbatical, I have taught that course every year for 22 years. Twice

[1] Dr. Jon Bentley is a Member of the Technical Staff, A.T.&T. Bell Laboratories. See Proceedings paper by Bentley.

I have team-taught it with David Parnas, which is phenomenally exciting. One year, I taught it with Bernard Witt, of IBM's Federal Systems Division, and another year with Constance Smith, who taught at Duke. The evolution of the course has been interesting[2].

I am teaching the software engineering project course this term, as for the last two years, over our statewide television network. I can see the students at the remote sites, and they can see me. That has an unexpected advantage: although I would normally be teaching live over the network, I can give them a videotaped lecture. That is what I am doing there this afternoon.

One of the laymen at the Microelectronics Center of North Carolina[3], who had been watching a lecture on a monitor installed in the lobby, said, "You don't seem to be teaching. You seem to be preaching." Indeed so. There are two reasons why. First, we do not really know that much to teach. Gordon Bradley[4] and Mary Shaw [5] in their papers speak about the lack of identification of principles. Second, we are trying to teach practices that we believe – and this is an article of faith – involve short-run pain for long-run benefit. Preaching always involves persuading people to undergo short-run discipline for long-run benefit. That is what preaching is all about. So it is no accident that a great deal of what we do when we teach software engineering is, in fact, exhortation. We are trying to motivate the will of the students, rather than merely to inform the mind. I expect some element of exhortation to be necessary forever. The conversion of our students' long-run ambitions into daily motivation is always an important function of the teacher.

Today, I want to talk about the state of software engineering as I understand it, and some opinions on the curricula issues that are before us. The viewpoint, I fear, will be that of the leper at the feast. After reading the pre-distributed position papers, I find I am in fundamental disagreement with a good deal of what is proposed, described, and practiced.

2. On Software Engineering

Engineering

I will start with my definition of software engineering. I like to distinguish four things: a

[2] The course handouts defining the step-by-step assignments and intermediate documents for the projects, and outlines of some of the lectures, are available as a UNC Technical Report. Write the author.

[3] The Microelectronics Center of North Carolina is located at Research Triangle Park, and serves six member institutions

[4] Prof. Gordon Bradley is a member of the Computer Science Department, Naval Postgraduate School, Monterey, CA. See Proceedings paper by Bradley.

[5] Dr. Mary Shaw is Chief Scientist at the Software Engineering Institute. See Proceedings paper by Shaw.

program, a programming system, a programming product, and a programming system product, [Figure 2]. Software engineering is concerned with building programming products and programming system products. In other words, it is proper to call it software *engineering*. It is indeed an engineering discipline – it focuses on building.

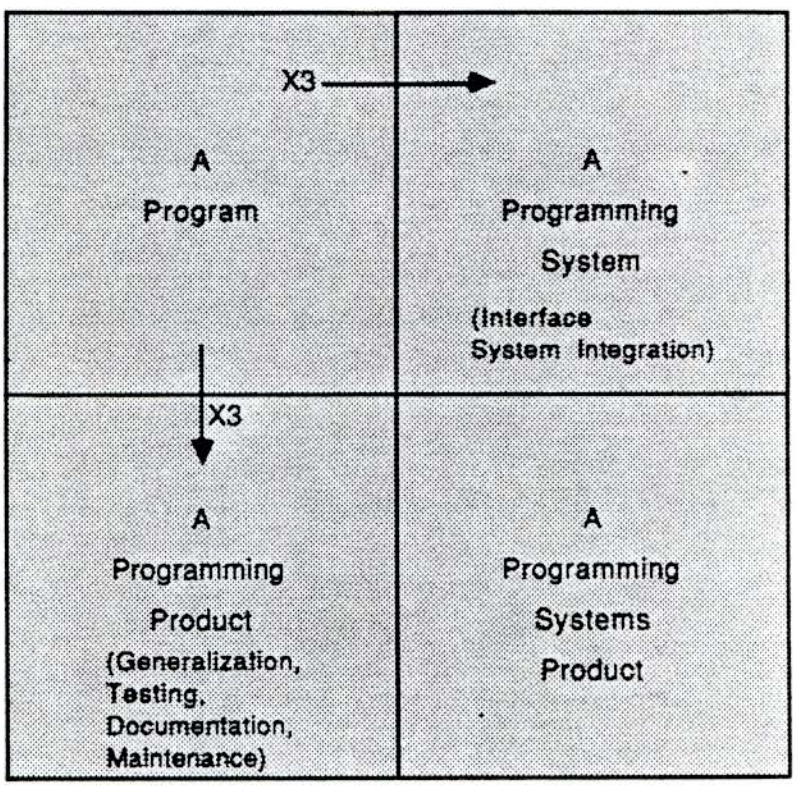

Figure 2

In graduate school I roomed with a high-energy physicist. He spent a year building the electronic apparatus for his experiments. He then spent two weeks at Brookhaven National Laboratory taking pictures of events in a cloud chamber, then a year looking at his 100,000 pictures. If you looked at the way he spent his time, you would have said he was doing engineering. On the other hand, I have known engineers who seem to spend most of their time taking measurements about hitherto unknown phenomena. If you were asked what he was, judging by how he spends his time, you might say, "He looks like a physicist."

The difference lies in the *motivation* and not in the *activity*. **The scientist builds in order to learn; the engineer learns in order to build.** That distinction we can accurately use to characterize software engineering. As an engineering discipline, it is concerned with quality, effectiveness, cost, and schedule – concepts that, if not alien, are at least of little concern to the underlying science.

Arbitrary Complexity

What is peculiar about the engineering of computer software objects? How does it differ from the classical disciplines? It differs in an important way from two of the supporting disciplines, mathematics and physics, from which electrical engineering derives. Most mathematicians and physicists dislike real-world computer science problems. The reason is that our problems are characterized by what I call *arbitrary complexity*. Anyone who has wrestled with an operating system and had to interface 44 different kinds of input-output devices; or a payroll system and had to deal with the income tax for 50 states, plus the federal government, plus innumerable cities that have peculiar income tax laws; or wrestled with the other forms of artifact that we have to build and the environments into which we have to build them, will recognize this as a common characteristic.

Mathematicians and physicists dislike this for different reasons. The mathematician dislikes complexity, and the mathematician's fundamental attack on complexity is to abstract. One forms an abstract model of the problem, solves the abstract model, and then applies the solution back to the original problem. That paradigm has been phenomenally successful. The history of applied mathematics, intertwined with the physical sciences for more than two centuries, is one of the rich results produced by that model. Increasingly, however, as one comes up against intrinsic complexity, we find that smooth models of classical mathematics do not work. So we come to fractal mathematics for describing or abstracting roughness. We continually have to invent new mathematics to deal with deeper levels of complexity.

On the other hand, physicists dislike the arbitrariness. They are no strangers to complexity. Anyone with 26 elementary particles recognizes that the world is complex. What they dislike is that it is arbitrary, because physicists, no matter how atheistic, are fundamentally convinced that there are *not* 26 elementary *anythings*; that there is a fundamental, unified theory to be found. It

is that faith that keeps the physicist going forward.

No such faith comforts the computer scientist. Our complexities are arbitrary, because they are the fruits of many independent minds acting independently. Consider the task of interfacing to an operating system 44 different input-output devices, each designed by a different engineering team. Unless there was a pre-existing interface, there is no reason to believe those designers acted under any unifying principle at all. This arbitrary complexity of interfaces characterizes much of what we do. It is a reason why we had to develop a new science, with approaches and techniques different from those of the classical disciplines.

What About Software Makes Its Engineering Hard?

A natural question is, "Does it have to be this hard?" Is it not just that we have not yet found the key to unlock the door? Studying the nature of these arbitrary complexities, we see that the essence of building software products is the complexity of the conceptual structures we are working with, rather than the labor of representing them. This complexity is compounded by the necessity to conform to an external environment that is *arbitrary*, *unadaptable*, and *ever-changing*.

If we ask ourselves, "How have the big gains in productivity and effectiveness in software engineering come in the past," I think we will see those gains – in high-level languages, time-sharing, unified programming environments – all broke major artificial roadblocks to *expressing* the complexities of our solutions and our problems. The high-level languages remove the artificial roadblock of coding programs in machine-level instructions in zeroes and ones. Time-sharing removed the artificial roadblock of limited access to hardware. The unified programming environments remove the artificial roadblocks that were caused by a lack of common file formats and command philosophies.

We will make progress by continuing to remove these artificial roadblocks, via workstations, better languages, richer programming environments, etc. I think, however, that fundamental progress can only come by really attacking the underlying complexity, not the difficulties of expression. There are many promising attacks, as Figure 3 suggests. I will not take the time to talk about them, because I really want to go on to curriculum. But I must remark that we vastly underestimate the work, the difficulty, and the error-proneness of setting system requirements in the first place.

Key ideas:

- Top-down design – N. Wirth
- Outside-in design, system architecture – G. Blaauw
- Incremental growing on an executable driver – H. Mills
- Information-hiding modules – D. Parnas
- Chief programmer teams – H. Mills
- Verification – E. Dijkstra, Floyd, Hoare
  useful, but limited by costliness
- GOTO-less programming – E. Dijkstra
  structure, yes; avoiding GOTO, no.
- Structured walk-throughs

Figure 3

Iterative Development is Crucial

I like Christopher Alexander's maxim in *Notes on the Synthesis of Form*: "The only way to define *fit*, is as the absence of *misfit*." If one wants to grind a steel plate flat, one takes an optically flat standard plate, paints the whole works with purple goo, slaps it up against the plate one is going to grind, then grinds all the purple places. Then one paints the optical flat with goo again,

slaps the two together again, and grinds the places that are still purple with a finer wheel until, finally, instead of *none* of it being purple, *all* of it is purple. So, the only operational way to define this optically flat plate, is as having no *bulges* or *valleys*. Correspondingly, I believe the process of dealing with arbitrary complexity, in terms of the user's requirements, is iterative: we build prototypes, put purple paint on them, slap them up against real users, and grind the places that are still purple – in the products, not in the users.

Iteration on a programming product specification is an inherent, proper part of the professional's job. We cannot stand back and gripe that the user didn't know what he wanted. We must take it as given that the user does not and cannot know what he wants about artifacts as complex as those we now build. The mind of man cannot imagine all the ramifications of such artifacts. There must be an iterative cycle in which the professional works with the user to define the requirements; demonstrates their consequences in human factors, cost, and performance; then in a prototyping phase iterates with the user to develop a product that is, in fact, satisfactory.

The Failure of the "Standard Software Development Process"

Let me offer a discouraging observation on the state of the art. I did a little mental study in which I wrote down a set of what I call "exciting software products." These are ones that have avid fan clubs, ones that people are crazy about. You can add names to this list, shown in Figure 4. We typically call the fans *bigots*: *APL bigots*, for instance. I think the ancestral group should be Fortran's. Those of us who work with physicists and chemists today, recognize that there are still Fortran bigots about! Each of these exciting products has such a group. I put Visicalc as the latest, but not the last.

I put a different set of things, which you can call the "work horses" of the field, in another category. This group is made up of things that are immensely useful, in many cases immensely successful, and have made major contributions to getting work done. People appreciate some of their successful characteristics and don't appreciate others, but it is very hard to find bigots, excited fans, about any of them. I have trouble finding any exciting software product – one that arouses passion on the part of its users – that was developed inside a normal product process. What does that tell us about the normal product process? About the state of the art? About the importance of teaching the normal product process? I think it tells us something about software products and designs in general: the thing that makes exciting software products is conceptual integrity, and conceptual integrity comes from individuals.

One can elaborate a little bit. Committee design is a minimax strategy. It limits the losses and goofs. It also limits the upper reach of quality, elegance, function, and speed. This is true of bridges, cars, movies, novels, paintings, music, etc. So the theorem I would leave you with, because I can't prove it, is that a product that *surely* excites *somebody* is more likely to excite a lot of people than a product that *more or less* suits *everybody*. The "work horses" I referred to, the

ones that do not have fan clubs, can be characterized as having "homogenized designs," and the ones with bigots, "idiosyncratic designs." The homogenized design process is aimed at producing products that more or less suit everybody. You may want to propose other candidates, and you might challenge some of my choices of candidates, but I think that the thrust of those two sets is unmistakable, [Figure 5].

3. On Software Engineering Curriculum

Standard vs. Individualistic

I think this theorem is also true of curricula. We may be richer, in the process of evolving a generally accepted software engineering curriculum, if we have a *lot* of places forming a *lot* of curricula and publishing them, than if we move too rapidly toward any kind of ***standard*** curriculum. If you look in many different college catalogs, you will see that there has developed a great deal of standardization among undergraduate physics curricula, for example. In the middle two years of undergraduate physics, one takes the same courses anywhere one goes, and one may take them from the same text books. Is this done through standard curriculum development by the American Physical Society? No. The similarity exists because the importance of the subject matter is self-evident: there is a consensus in the field of what the principles are. I suggest that a standard curriculum be grown organically by developing a set of principles. That is the only way to make it durable, important, and portable.

Does that mean it is not useful to develop model curricula? Of course it is useful. In any branch of art, the people who went through it first, and learned what not to do, can be of great service to those who come on the scene by explaining where the pitfalls and minefields are. Sharing experience with curriculum development saves people from making the same mistakes again.

The most important principle to teach a software engineer is, "Don't build software (if you can help it)." It is almost always cheaper to buy it if you can, and it is almost always cheaper to buy it even if its price is about the same as your estimated cost for you to build it. That is, one generally underestimates the effort required to build product-quality software. Even by buying it, you may not get product-quality software, but your odds are much better.

Exciting Software Products

Outside Product Houses	From Product Houses
Fortran	
APL	OS/360
COBOL	
Pascal	Algol
LISP	DEC's VMS
C	PL/1
UNIX	Ada
Tenex	IMS
Visicalc	
VM-CMS	
System R	

Figure 4

Committee Design is a Minimax Strategy

- Limits losses and goofs
- Also limits the **upper** reach of

  quality/elegance
  function
  speed

- Bridges, cars, movies, novels, theorems,
  paintings, music
- Idiosyncratic vs. Homogenized

  A product that **surely** excites somebody is more likely
  to excite
  a lot of people

  than one that **more or less** suits everybody.

- True of software systems
- Of software engineering curricula, too.

Figure 5

Permanent vs. Transient Truths

From the perspective of looking at seven computer revolutions over the past 40 years, the first thing that strikes me is that one has happened about every six years. Second, most of what we learned and talked about in the 1950's, we would not think of teaching today. Much of what we taught is no longer true, or if true, no longer relevant. Are we training people for an initial job or educating them for a career? If we are educating for a career, I wholeheartedly support Mary Shaw's identification from the Carnegie Plan of what is involved in professional education for a career. We need to teach them to think like software engineers, rather than to train them in 27 programming languages, 15 methodologies, and 30 tools. That means they will have to be exposed to some methodologies, some tools, and some programming languages. But those are not our *objectives*. Our objectives are to shape ways of thinking, and, by experience at wielding some tools, to develop and facilitate the implementation of new tools in the field.

That brings me to the points about which I would argue. It seems to me that all the central questions about software engineering curricula can be summarized by a set of dichotomies, as in Figure 6.

Thin vs. Fat

Let me put forth another theorem: if you do not know what to teach in a software engineering curriculum and, if in putting one together, you find a lot of modules that are short on principles – where one can teach only tools or methodologies, or today's practices – instead of most of those modules, teach nothing at all. Instead, encourage students to spend those hours learning something such as physics, mathematics, or accounting, which they *do* know what to teach. One of the most valuable courses I had as an undergraduate, and today use regularly, was a one-semester course in accounting for non-accountants.

I would not offer an undergraduate software engineering curriculum at all. I would offer undergraduates a two-semester software engineering course, as part of a computer science curriculum. And I would recommend an undergraduate computer science curriculum only to those planning to stop with the B.S. Young people come to me and say, "I want to be a computer professional." I reply, "Do you want to go to graduate school, and become a real professional?" If they say "yes," I say, "Do not take a computer science major as your undergraduate. Get educated."

Our oldest son fell into that fiery passion for computers which often strikes in the teen years. It is very much like being engaged and being married. You want to experience and enjoy that initial passion, but you would like to grow out of it into a more mature relationship, one that will always be fired with moments of the passion. I encouraged each of our children to do that with the computer passion while they were in high school, because it can ruin a college year if it first strikes then.

Software Engineering Curricula

Standard	vs.	Individualistic
Transient	vs.	Permanent
Fat	vs.	Thin
Narrow	vs.	Broad
Hollow	vs.	Solid
B.S.	vs.	M.S.
Science	vs.	Design
Projects	vs.	Exercises

Figure 6

When that son got ready to go to college, he wanted to study computer science. I said "Well, if you really want to work with computers, do a physics major. Study all of the sciences. Do not fall into this one just because it is handy." (He had been exposed to a lot of computers during his life.) "Then, when you are a senior, if you still want to become a computer scientist, I will quit hindering and start helping. But first, sample all of the sciences to see if your infatuation with computers comes merely that from propinquity." The summer after his junior year we were walking on the beach and I said, "Well, son, what do you think? Which subject interests you most?" And he said, "You go into a room, and you look around, and you say, 'Look at all the pretty girls.' Then

you say, 'But *this* is the one I love.'" I said, "I quit." He is now in a Ph.D. program in computer science at Stanford.

Look at your undergraduate college experience. Which parts do you retain as most valuable? For me, it is a Shakespeare course, a French literature course, a lot of experience in public speaking, a lot of training – extracurricular and curricular – in how to run meetings, writing training, an accounting course, and especially some courses in electricity and magnetism. Those experiences are still very useful to me. I can tell you lots of things I spent hours on that I have not used. Some of them were in the liberal arts, but many others were in the major.

In like manner, we must train professionals who have been educated to be citizens, leaders, and communicators. The software product today consists of more documentation than code, and the good software product today includes *good* documentation. How will you learn to write if you have not studied the good models of writing and practiced the techniques? Do we want to displace a broad and useful undergraduate education with training in software engineering tools and methods? Surely not!

Broad vs. Narrow

At the graduate level means I would recommend a particularized *software engineering* curriculum only to practitioners who have had field experience and are coming for career upgrade education, who know they are getting a specialized, technical training, not graduate education. For all new software engineers, I would recommend a master's in computer science, with several courses in software engineering, but not a software engineering curriculum. Why? Because much of what we teach today will not be true ten years from now, and a great deal of the rest will not be relevant. More important, they will *need* the broader knowledge.

Can anyone in the software-building business really operate without understanding simple accounting? Can anyone in software engineering really operate without understanding the principles of at least the first course in numerical analysis: concepts, error propagation, and the vagaries of floating point?

Most curricula being put forth for software engineering – and there are exceptions in the Proceedings – give a one-course, at most, discussion of computing machines. Undergraduate exposure, which typically is one course in machine architecture, is assumed for the graduate courses – and those may include another one. Are we going to build all of this software without understanding the engines with which it will run, the trends shaping those engines, and the ability to project the corresponding advances for hardware that will revolutionize the kind of software we have to build? So, I would argue very strongly for broad vs. narrow.

Solid vs. Hollow

All the university departments I know want to create a lot of courses that address topics at the very forefront of the field. Why? Because that is what the faculty wants to teach. So we construct, particularly at the graduate level, what I call "hollow curriculum": in football terms, no blocking, no tackling, but Statue-of-Liberty plays all over the place. We shuffle the core curriculum courses off to the most junior faculty members to teach, and we elders teach the advanced ones. A solid curriculum is one in which those intermediate-level things that seem like old hat to us, but are not old hat to the students, fill in the interior. These are the established principles represented by algorithms, data structures, operating systems, languages, machines, and compilers.

Design vs. Science

The science vs. design debate rages in engineering schools everywhere, all the time. I think the papers in the Proceedings properly emphasize that if the motivation is to build, we have to teach the art of design and not merely the supporting sciences. It is in this respect that software engineering courses differ from many of the underlying computer science courses. We must teach people to design. The only way to teach people to design is to have them design, criticize, have them redesign, and then to have them build the designs.

Projects vs. Exercise

What they design brings us to the exercises vs. projects question: How little and how big? I think the answer is you really want to do some of both. The real issue is the precise balance between exercises and projects.

Great Designers

Now one more thought: If we go back to the list of great software products, exciting ones, we observe that they have a conceptual integrity that comes from very small design teams: Fortran with half a dozen people, APL with two people, and so on. These design teams are not only small, but also comprise really first class minds.

Let me suggest one more principle: **Great designs come from great designers**. Good designs, as distinguished from bad designs, can be produced by teaching people good principles and proper methods. We take the step from good designs to great designs, however, by finding the people who have the talent to do the great designs.

Look over the whole body of classical music. How much has survived? On any classical music station you can hear obscure selections from the 16th to the 19th centuries. When you listen to them, you know why they are obscure. You can count on your ten fingers the great composers in each century. Indeed, even they have written some losers, but there really is a qualitative difference between the great and the obscure.

That is true of bridges, and that is true of software products. So an important part of our emphasis on teaching must be to identify and to equip, by special means if necessary, those who will be the great designers. We should make it part of our business to find them, and then to provide whatever is necessary in education, nurturing, sheltering, and diversified experience, to enable them to make, their contributions in their turn.

One last comment about great designers: I am not one, but I have spent a lifetime trying to find them and enable and focus their efforts. That, too, is fun, and an immensely satisfying part of the teacher's reward.