Talk given at Software Engineering Conference Atlanta Georgia. (ex tempore) I thought it wd be 20 min

but never written up

~~and the speaker agrees after he sees what he did see in print, it will be published in Computer Magazine. Those of you who don't get Computer Magazine should stop at the registration desk about joining the Computer Society.~~

Our keynote speaker ~~at the present time~~ is Professor of Computation at Oxford University. He works in the Computer Laboratory; at the head of ~~works in~~ the Programming Research Group. This is ~~another~~ a recent move for Tony Hoare, ~~those of you who have known him mostly at his previous job which he held for~~ who spent nine years as Professor of Computer Science, The Queen's ~~Kings~~ University, Belfast. It is my pleasure to introduce Professor Hoare

New speaker

This talk will be ~~is~~ rather like a sermon. ~~perhaps.~~ And like a ~~good~~ sermon it should start with a good text. The text that I have chosen to start with is a quotation ~~which I'm afraid I haven't been able to trace directly,~~ attributed to Thomas Tredgold, the President of the Institute for Civil Engineers, in 1828. He defined civil engineering as "the art of directing the great sources of power in nature to the use and convenience of man." I would like to adapt that text to our own particular branch ~~topic~~ of Software Engineering, replacing "the great sources of power in nature" by ~~the phrase~~ "the great computational power of the electronic digital computer." The reason why I have chosen this text is that ~~one~~ I believe that the directions in which we have been working~~, in which I have been working,~~ for a number of years, although very necessary and very desirable, are not the whole of engineering — that ~~that~~ we are still only after many years ~~9 months~~ exploring only the foothills of what we hope will one day become a fully fledged engineering ~~subject.~~ discipline.

And so I would like to supplement the remarks of ~~not so much as to contradict~~ our general chairman ~~as~~ ~~to add~~ ~~to his remarks~~, to point out the danger that ~~of~~ our (present) studies are becoming ~~maybe getting to be a little~~ too introverted.

We are studying the problems of programming, the practices of programming, the management of programming, structured programming, (programming languages), programming tactics, program proofing, ~~they all have~~ - all these topics are wholly relevant, very necessary and very important. But the overall "objective" that we must all seek is not some "perfection" in our programming methods, but rather the usefulness and the convenience of our programs to our fellow man. ~~precepts but the objective of our programming so the computers on which they run shall be~~ ~~of service~~ ~~assertive to fellow beings. I would like to another thing which I might reconstruct as the early history of engineering~~

I would suggest that our present achievements in Software Engineering do not yet bear comparison with the present status of longer established branches of Engineering; rather we should go to

~~Tape #1, Side 2~~

~~Emerger of science and practicalities of which it's not.~~ Back to the days of the master craftsman. The master craftsman, - the shipwright ~~chip rights~~, the ~~cathedral~~ builder ~~procedural buildup~~, the joiner, the mason, - was ~~although~~ a remarkable genius ~~expedience~~, with highly primitive tools, and with little ~~liberal~~ theoretical understanding, he somehow managed to ~~build~~ construct ~~a~~ products, ~~buildings,~~ cathedrals, ships, chairs and tables, bridges, — which were useful, and convenient, ~~sturdy~~, robust and even elegant or ~~and useful~~ ~~or spectacular. I don't know if any of you familiar with the Saint Chapelle in ~~of~~ Paris. It has fantastic walls of glass and stone faces. And how's the craftsman who put that up ever thought that he would sell her, I just don't know.~~

Consider now ~~What was~~ the transition between the day of the master craftsman and that of the engineer ~~here today~~? I would like to suggest that the transition occurred when the craftsman (first) started to use pencil and paper to plan his designs in advance; to make his ~~story~~ ~~professionally~~ ~~xxxxxxxxxx~~ drawings sufficiently accurate and precise and ~~yet~~, of course, ~~abstract so~~ that they could determine ~~the process for~~

the details of the construction of the product. He measured the distances on the line, he drew and redrew the moving parts until he was sure ~~that they would not fight against each other~~, tax ?? that they were free to move in the right way. He then ~~brought~~ got these designs drawn up, even more carefully ~~actually,~~ by professional draftsman, whose task it was to exercise meticulous accuracy over long periods of time. The ~~designs~~ drawings were checked by checkers who spent an equal amount of time making sure that they were all accurate, that they all ~~stick~~ fitted together, and ~~designed~~ described a product uniquely and completely. And then these designs were passed on to quantity surveyors ~~to the producers, the design for them to use, well first of all shall we say,~~ to get an accurate and complete estimate of the ~~life~~ likely costs of the product. The number of components, the number of bricks required, the number of windows , the number of window panes, — all of these ~~things~~ can be computed in advance on the basis of a precise and accurate set of drawings and plans. And then the design can be handed over to ~~a~~ production teams ~~of people~~, each of ~~whom~~ which is responsible for implementing a small part. This may take time. It may raise many problems.

But if each (member of each team is) ~~feels~~ responsible (for ensuring) ~~that~~ his part is done properly, it is ~~nice for~~ known that when those parts are put together, the whole construction will work. This is because the design has ensured that they will. There is nothing more that can go wrong. ~~Except, with of course, in engineering there always is.~~ If you do find that the putting together of these independently constructed parts is taking any significant proportion of the scheduled time of delivery of your product, then who is to blame? It is almost certainly the engineer who did the original design. It is he that ~~He~~ made them inconsistent or ~~he made them~~ incomplete. ~~and~~

So I think we're quite right in our research and (in our practice) ~~practical approach to~~ of Software Engineering to concentrate on the ~~old~~ whole process of making rigorous designs of all our software products before we ~~put~~ get them ~~out to be~~ implemented on a particular piece of hardware or in a particular programming ~~run~~ language. And we should begin to insist that these designs are accurate and complete, and even voluminous ~~and they shouldn't be leaving us~~ before the task of ~~loading~~ coding starts. ~~Then~~ Only in this way can we be sure that the product will ~~stick~~ fit together when it is ~~put together~~ eventually assembled.

A product in which the management plans to spend over half of ~~name of~~ ~~xxxxxxx science~~ ~~can't~~ ~~xxx overcome~~ the total efforts ~~of the project~~ in ~~xxxxxxxxxxxx~~ integrating or integrated testing of components which have been separately constructed is a project that has been incorrectly engineered from the start. ~~and we know several of those~~.

~~Now~~ The drawing practices which ~~knowing of the facts is which~~ evolved from the needs of engineers to make rigorous designs of their products, have been ~~evolved and~~ developed and formalized in a~~n extremely~~ rigorous fashion. ~~and it carries the~~ Standard codes of practice ~~and~~ ~~breakdown~~ ~~laydown by~~ ~~that~~ have been laid down by the institutes of engineering and are meticulously ~~may not be~~ followed by every practicing craftsman and engineer. If an engineer does not follow these practices and his ~~work~~ product fails to work, ~~or~~ he can be sued. If his product ~~breaks or~~ kills someone, there ~~is~~ will be an inquiry and he will be barred ~~drummed~~ from the practice of his ~~dropped out of the~~ profession. Perhaps engineers of the past ~~are a little bit~~ were more fortunate than programmers, ~~but~~ for when their product fails, people do get killed; and therefore, people take the business of proper engineering more seriously in bridge building than they do in programming.

~~Now~~ The use of blueprints and scale diagrams involves a little bit of mathematics, even mathematical theories. We need to be able to measure, ~~we need to be able~~ to count, ~~we need to be able~~ and to multiply. ~~and~~ We need a little knowledge about similar triangles and Euclidean geometry in general. ~~How to draw elipses, and conics~~ ~~so on, is necessary in the practice of the job of a draftsman.~~

and the methods based upon them

~~Now~~ These theories (now) are taught in quite elementary schools and ~~they are taught~~ in technical colleges ~~authorities~~ a ~~and~~ They are ~~very~~ widely practiced, and it is generally accepted that a formal ~~ity~~ of theoretical understanding of the underlying principles of geometry is required of the draftsman, as well a considerable skill in practicing these principles ~~skills~~ in his drawings. I would like to ~~might~~ suggest that we have found an analogous ~~the same~~ phenomenon in programming. Here, unfortunately, the formalism ~~vocalisim~~ that we need in order to specify ~~satisfy~~ and control ~~confirm~~ our programming practice ~~own processes~~ is not pictorial. ~~has not been good~~. It is a logical formalism, rather ~~although~~ an austere ~~design~~ formalism involving the more elementary parts of mathematics, arithmetic, and mathematical ~~formal~~ logic. Of course, we can draw pictures but, ~~approximately in general, except in~~ as examples, ~~shall we say,~~ instructional ~~kindergarten supplement material, pictures are not acceptable~~ serious design work ~~to use for graphs~~. ~~The important part is the linear logical paths and specifications.~~ ~~I don't think that we should expect~~

~~very logical in linear and maybe even~~

only as an illustration, to confirm an understanding which has been conveyed by

I don't think that we can ~~necessarily~~ expect to ~~imp~~ use ~~have~~ a fully automated ~~open-minded computerized~~ design ~~ed~~ process when we are producing our programs. I think that if we try to bypass the phase of manual production by enlisting the aid ~~s~~ of computer too early, we run a severe risk of sinking under the inflexibility of our compilers, and surrounding systems. ~~or whatever~~. We ~~may~~ will have to ~~i~~ evolve manual practices in the same way as every other engineer in the beginning, ~~would begin~~, before we can ~~xxxxx~~ determine exactly /which part of the process can be automated. ~~Our~~ I draw an analogy here with the ~~xx~~ relatively new disicipline of electronic engineering and electronic system design.

A logic designer uses logic; by logic here, I mean abstract logic; logic that is studied by magicians ?? ~~mathematicians~~. And he has to formalise ~~he has~~ a complete design of his machine ~~business scene~~ in pure logic with no physical components in it at all. ~~And that logic design~~ is ~~appears as it were,~~ a complete process of ~~the functional~~ specification ~~components~~ ~~of his machine.~~ ~~And he gets~~ In principle it would be possible for a computer program to take that logic design, assign positions to every required electronic component in it, lay out those positions on ~~shall we say~~ the surface of a chip of silicon; ~~in tracking~~ to track connections ~~wire~~ between the components, across many levels, — in other words, ~~and there we have~~ a fully automated, integrated, circuit production system. But it doesn't work! It doesn't work because there is no way of automating the transition between the specification of what a product ~~component~~ is to do and the way in which it does it. There is a great deal more engineering skill required to transform the formal specification to a workable design which can be photographed onto a piece of silicon of acceptable size.

new para

~~And~~ This two-stage process may be something that we as programmers should not be unwilling to copy. We should be willing to write a program formally in all ~~the~~ some very higher level language, ~~that we can imagine~~, a language which is so precise and, I hope, so concise that it is possible later to transcribe it, to implement it in many different ways in languages of a very much lower level; these languages which are designed to specify~~ing~~ details of ~~feed panels, on~~ how the machine is supposed to execute the program rather than the general function which ~~the details of what~~ the program is to carry out. ~~do.~~ ~~And I think that tiring?~~ We should be willing to carry out ~~do~~ this program transition ~~counting processing~~ ourselves. ~~I think that maybe,~~ If I can press ~~use~~ the analogy a bit further, we have not yet discovered the discipline of logic design. The designers of ~~our~~ programs of the present day are like designers of computers who have not yet discovered that you can abstract from the positions of logic components and the tracking and the back wiring of ~~the~~ computers, and Who ~~you can~~ conduct the whole designs ?schedules ~~tribunes~~ of incredible lengths. ~~And our~~ Yet the progamming languages that we actually use ~~now~~ require us to ~~satisfy~~ specify our problems in that sort of degree of detail. ~~Now~~ Such engineers would find ~~it is~~ it extremely difficult, as you can imagine, to ~~guess~~ get the whole design of a computer correct, It is so difficult that we would never have built computers if they had had to be designed in this way. ~~And yet, of course,~~ Unfortunately, this issue of ~~this~~ programming language ~~design~~ is so fraught with political and commercial consideration that ~~very often~~ ~~when you~~ on starting a project the most important decision, that is often ~~to be~~ made by the executive president of the whole company is that you shall program in some particular ~~that~~ language. And so the poor programmer is faced with the task of thinking right from the beginning in terms of that language. He writes little bits of code and then he tries to put them together. Of course, since he is not working to a completed design, he has the gravest difficulty ~~That, I think, is only the beginning of the things not to do. So in the programming languages it is one of the few branches of software engineering to which I have given some thought.~~

~~At present, programming languages, the ones that we use,~~ ~~have largely been designed not by engineers but by some, what I might call, random political and historical process. And I have seen many of them.~~

~~With general engineering,~~ Earlier, I suggested that an appropriate analogy for the position which we are in now ~~would be with the~~ is at the transition between the craftsman and the engineer. In programming language design I'm afraid we have hardly got to the ~~confident~~ craftsman stage; ~~of~~ the nearest analogy that I can think of is the alchemist. The alchemist was seeking a very worthwhile goal, the synthesis of gold. ~~so we call it a~~ Similarly, our political masters are asking us to discover a programming language that actually makes programming easy.

Obviously, it is worth spending a great deal of time and a great deal of effort ~~for~~ on achieving such a goal. The king who manages to discover the *philosopher's* ~~S~~stone, the ingredient that will turn basemetal into gold is obviously going to have no difficulty in paying his army.

And so, many learned people sought the magic formula for producing gold. Many experiments were made. Some people thought it was ~~the~~ some exact mixture of well-known ingredients that ~~you would have to find~~ must be sought. And so, we see many languages being developed which are, shall we say, something of a rehash of well-known features in a slightly different ~~texture~~ mixture from before.

Other people are still seeking some sort of magic ingredient which only if they can find and put it into their language, all will be well. Now what happen~~s~~ed? After while people learned ~~xxxxx~~that maybe ~~this was~~ making gold is a difficult thing to do; ~~they couldn't~~ and if you can't make gold yet, ~~but~~ nevertheless it ~~was~~ is worthwhile studying the chemical properties of matter. And so, we ~~saw a~~ see the transi~~s~~tion between the alchemist and the chemist. And after a while the chemist discovered a number of scientific laws from which they could deduce that the manufacture of gold from base metals ~~was~~ is after all impossible; physically impossible.

I suggest ~~that~~ maybe if we study the logic of programming of computers with ~~specific~~ particular care, we can see from ~~the~~ its very nature ~~of the factor itself~~ practice that it programming is relevant to the users ~~as it is~~ it is always going to be an art ~~always a thought~~ requiring the ~~focus~~ fullest attention of the most brilliant ~~men~~ engineers, right from the point of conception and design down to the details of the implementation in octal, ~~shall we say, on a computer.~~ in some code which is readable and executable by computer.

19.

To return to the analogy with the history of chemistry

In this century ~~A little later on, of course,~~ the physicist came along and ~~said~~ found that it was possible to produce gold by breaking up the structure of the atoms of the metal~~;~~, but even though it was possible, it was vastly ~~very~~ more expensive ~~and was at least 150 times as expensive~~ to make gold ~~as~~ (than simply) ~~to~~ dig it out of the ground. And that economic fact is an even more impenetrable barrier than the physical ~~side~~ one! ~~Eventually we find that we can do it! It can be done! But, its just not worth doing. That sort of project is one that we should try to avoid.~~

So at the moment, many scientists are seeking methods for ~~[illegible]~~ [removing] the skill and judgement required in programming. They work on automatic program synthesis and verification. At present, it seems that what they are doing is impossible. But even if they succeed, they may find that it requires even more skill and judgement to use their automated systems than to write the program in the first place.

~~Where~~ The design of programming languages is obviously one of the ~~abstracts~~ aspects of software engineering in which good engineering principles should be employed, especially since ~~It is also the product of programming language, or~~ it is a programming language that is ~~are~~ the basic tools of ~~the~~ other software engineers. ~~This is an area where~~ So an understanding of software engineering, as applied to other application areas, is (also) absolutely necessary. We must understand the programming process, both in its logical and its sociological aspects before we can design good programming languages.

Now I would like to ~~I would like to~~ take a little time to suggest, ~~maybe~~, how this ~~can~~ may be done. And the first hypothesis I will put forward for you is that the direction of ~~the of~~ advance in programming language design is backwards. We need ~~a term of~~ to turn (of our) research and development into reverse. ~~At present,~~ For the last ~~ten years~~ decade we ~~take~~ have taken it for granted that the programming language which has more features in it must be better than one that has less. The obvious ~~thing to do~~ improvement, if you see a good programming language, is to extend it. (laughter) I would suggest that exactly the reverse is more appropriate: (we should contract our languages to improve them.) ~~It is a very difficult task to do. I will explain a little bit~~ more ~~why it is difficult and why I think it's worthwhile.~~ ~~But~~ And there is some evidence for this hypothesis.

~~There are some far worse languages in which people write programs than we have available~~ . People write programs in machine code. And the most surprising people do it. Hobbyists, with the most curious little machine code, ~~really quite complicated,~~ manage to do it. I have even known managing directors who programmed their own personal computers in machine code. Hardware designers who design micro codes have an ~~even worse,~~ even more horrible logical tool in which to work. ~~I guess~~ And yet they ~~could use~~ produce thousands of words of correct micro~~specs~~ code, ~~and of course, scattered through it are a few words of incorrect~~. The reason why they can do it, ~~or~~ I would like to ~~do it~~ suggest, is that the tools or languages, they are using, ~~voluminous as it~~ ~~all the words may be~~, ~~it is~~ are simple enough for them to understand in ~~its~~ entirety. They (always) know in principle, they ~~sometimes forget, of course, but in principle they know,~~ and they know that they know, the effect of everything that they write down on a piece of paper. ~~It is a clear sacrifice; it may be complicated, but it~~ situation Each instruction they write is independent of every other instruction in the machine. The interaction effect between various instructions is confined to changing ?? one of a ?? small, ?? number of registers; and ~~is confined to no registered ? , the number of registered and the number of spaces, 54,~~ their language is sufficiently small and ~~irregular~~ that they can ~~'t~~ keep ~~control of~~ the whole structure in their ~~little~~ heads at one time.

And this is why one can accomplish great feats of engineering with very simple languages even though they are at a very low level. With a complicated language, the programmer is immediately taken out of control of what he is doing. He is never quite sure what the effect of what he is writing is going to be. And the only way for him to find out it to look it up in a manuals ~~that thick~~ several inches thick, or try it on the machine, ~~or~~ and he nearly always chooses the second of the two, the experimental method. And this is a very bad attitude towards programming. Programming in principle should be an entirely predictable exercise. If, as programmers, we take the responsibility (which we must do as engineers), for the correctness of our designs, then we must have tools which we can understand and keep wholly in our heads as new para ~~Now~~ I think it is extremely difficult to design very small languages; however, when they are designed I would suspect they will be surprisingly easy to use. I would like to suggest that the difficulty of designing very small languages, ~~so I'll call~~ them micro-languages, it's due to the fact that such a specification must be based ~~on an~~ in flash of insight, a discovery of roughly equal intellectual difficulty as the discovery of the simple laws which underly the processes of nature.

Without being arrogant, I hope, let us take the example of the laws of motion. I'm told that Kepler ~~went~~ put forward three laws of motion and this was a very great improvement over the tables of the motion of the planets which had been constructed by ~~the~~ Tycho Brahe a bit earlier. He had compressed ~~the~~ an enormous amount of apparently random complexity into just three laws. Newton who came afterwords did not say "~~Oh, goody. Three laws;~~ That is an excellent theory, but obviously insufficient. Let's ~~make some more~~ add some more laws". He managed to make progress in the reverse direction, and reduce the number of laws to one. So I would like to suggest that in doing research on programming languages ~~and software~~ and this ~~abstract~~ aspect of software engineering, we should try to discover ~~think~~ how small a language is adequate for our purposes. This is a matter of genuine research. We should get ~~our~~ a research students not to design new languages with ~~cobble together some anthology of main features~~ as many features as possible and then see if they can implement it ~~before~~ within 53 yrs. ~~is passed~~. That's of course a difficult challenge. But rather the reverse; ~~Nevertheless~~ ~~they~~ he should throw out as much as ~~they~~ he possibly can from his favourite language, and then show
(End of tape)
that the resulting language is both usable and useful. And this can be achieved by the normal scientific method, by experiment. Let us just try designing

see whether it hurts or not

~~Tape 2, Side A~~

These experiments, like all experiments, should have been done long before. But it's not too late now. In ~~Let us see what we can throw up;~~ we know certain things prove ~~are~~ difficult. We know ~~the chances~~ jumps are very difficult. We know references are ~~lacking~~ tricky. But what would happen if we actually took our courage in our hands and threw them out of our languages! We can't do that in ~~in this~~ a practical programming environment; project — it would be too risky. ~~it's not safe~~. But as an experiment, ~~And maybe indeed they are.~~ let us explore how far we can get in programming large and difficult problems using small and simple languages. I would suggest maybe you'll get a lot further than programming them in large and difficult languages. For simple problems, large and difficult languages are good enough. The complicated ones, impossible. So, let ~~us have a few Nobel prizes.~~ me throw out a few challenges. Who is going to be the first person who writes ~~an~~ a large operating system without a dump? ~~Well~~ that would be worthy of a Nobel prize. The first person who writes a large compiler without any references? All these pinnacles ~~or debths~~ are waiting to be ~~found~~. climbed.

¶ ..ll, how are we going to make our languages simple. Of course, there are two ~~ways of doing it.~~ basic approaches; ~~And~~ I would like to see research in both directions. ~~Once~~ Firstly we can make our languages more specialized towards ~~the~~ specific problem areas, ~~W~~we can use some of the understanding that we gained in the design of general purpose languages, some of the insights into the logical nature of programming and its mathematical foundations, into the ~~receptive~~ necessity of ~~security~~ strict typing and ~~thoughts and maybe even the avoidance of jumps.~~ and so on ~~We can transfer~~ and apply that understanding to more specialized languages, specialized (for ~~shall we say~~ example) to statistical applications, ~~special~~ social sciences, symbol manipulation, matrix calculations. All these special areas. Rea~~dy~~lly needs ~~of~~ for specialized languages with concepts built in which are ~~principally~~ appropriate for that particular application area. There ~~I think there could be~~ is a need for a family of such specialised languages, ~~in this area.~~ which should as far as possible be conceptually and notationally consistent. ~~I would like to see them somewhat~~ ~~It would be convenient if they all~~ Certainly, they should all use~~d~~ the same control structure, the same ways of defining functions, and parameters, etc. ~~But~~ they should be differentiated,

In addition to eliminating features which are believed to be of doubtful safety and usefulness, how else

~~Well~~ just in the same way as we have many different hand-held calculators. We have calculators for navigators, calculators for business applications, for hotel applications, they all have specialized keys, ~~certainly this is a number of keys and~~ but there is no reason why they couldn't have a reasonably standardized keyboard layout ~~with p ?~~ peak? and a reasonably standardized display.

The other thing that we can do (to simplify a language) is to specialize its level of abstraction ~~to level.~~ There are such things as high level languages and there are such things as low level languages. ~~And~~ Sometimes we need to use one and sometimes we need to use another. Sometimes we need to use both, and make a consistent transition between ~~those~~ them. If we could separate out the high level ideas and low level ideas a bit more successfully, I think we would get to see a great simplification in our programming languages. The high level ideas are ~~I like to think of~~ those derived from elementary pure mathematics, for example, ~~concepts,~~ numbers, operators, ~~functions~~ sequences, mapping and functions. ~~These~~ The low level ideas are derived from the ~~structure,~~ historical structure of ~~the~~ our computers, and include ~~things like~~ registers, addresses, jumps ~~jobs~~, condition codes, and other necessary ~~thoughts~~ but more intricate details of machine architecture. When we are originally thinking about our problems, I think we should use very high level languages, even though they cannot be implemented by any reasonable compiler. ~~and then~~ When we are addressing our computer, there is no reason why we should not use, if necessary, a ~~rather low~~ much lower level language, a ~~A very low level,~~ machine-dependent language perhaps. There is a large gap ~~We have a gap you see~~ between a language like PL360, or BCPL, (or even ~~just about~~ assembly code) ~~.~~ So ~~And~~ a language like Pascal which contains several useful ~~is almost as powerful as~~ mathematical concepts; and there are a few languages like Gypsy which are at a higher level than PASCAL. ~~Of course things go complicated because they put all the others more detail things in as well, on the principle of course, you make a general purpose language by putting together everything that had ever been thought of before. Maybe by keeping the levels very separate, we can~~

Now it might seem useful to construct a language as a mixture of features from all these different languages. Unfortunately, in practice, such a design gets rapidly very complicated, not only because it has a lot of features, but because there are a lot of subtle but dangerous interaction effects between features at all the different levels. The successful user of the language must understand not only the high level and the low level features, but also the way in which they are related to each other in a particular implementation of the language. It might well be simpler for a programmer to use two separate but simple languages: first a pure high-level language for designing his programs and then a pure low-level language for implementing them; and he should expect to make an efficient transition ~~by~~ between the two languages by hand and eye, not by automatic translation

Now I would like to explain why I think

~~the levels, there~~ is ~~is also~~ where the complexity and unpredictability ~~of modern languages arise. So as an engineering product which~~ ~~have~~ s ~~been designed, I think according to engineering principles,~~ ~~I would like to put a small plug for~~ Pascal. PASCAL has been such a technical success. Like every engineering product this is a compromise. It is a compromise that has both high level and low level features in it. It can be used in an abstract way and in a fairly concrete way. Of course, it doesn't give you as much control over machine details as more machine oriented languages like ~~lest~~ JOVIAL. ~~might list.~~ Of course it doesn't give you the same ~~peak~~ facilities for abstraction as a really high level language like Gypsy. ~~Cadell?~~ ~~, but~~ It is a compromise between these two directions. ~~things.~~ Like all compromises, it can be criticized from ~~in~~ all sides. ~~directions.~~ But like all well engineered products, it is extremely difficult to improve. Every detail of the compromise has ~~to~~ been thought out by exploring carefully the entire design space around that particular compromise. ~~I would like to say that~~ Even the defects of the language have been designed with the same care as its merits.

~~Let me tell a little story.~~ There's a large company, which about 5 years ago, discovered that one of its software writing divisions was using PASCAL ~~Pastel~~ for writing in software whereas another division had invented a language of their own which shall be nameless. This of course offends management's susceptibilities. ~~acceptabilities~~ sense of tidiness: ~~and~~ They thought that the software image ~~impact~~ ~~of the software~~ Company would be greatly improved if they could have both software writing divisions ~~agencies~~ writing in the same language. So they set up a committee with equal number of representatives from both divisions to decide which language to use. ~~And~~ Three months later the committee reported that there were seven votes in favor of Pascal ~~golf~~ and seven in favor of the other language. So the management reconvened the committee and gave them ~~said we will give you~~ another three months to ~~and you'll~~ design a (new) language ~~of your own which will be~~ as an amalgamation of the best features of both the rival ~~your two~~ languages.

(Incidentally the design of PASCAL took perhaps

Three months later they came up with a report which was, ~~shall we say,~~ ~~loudly~~ avowedly incomplete, inconsistent and incomprehensible. ~~So~~ They were given another three months to tidy it up a bit and then get on with the implementation. ~~of course,~~ That was five years ago, and as far as I know, ~~to cut a long story short,~~ nothing has been implemented, nothing has been decided, they are still designing. As indeed one might have predicted. It is very difficult to improve Pascal. I would suggest that even the original author of ~~that although~~ the language himself when he moved from the original Pascal to the Mark II version, in fact made ~~makes no improvement at all,~~ no measurable improvement.

The reason is ~~it~~ that PASCAL is a compromise. You do not improve a compromise by asking it to meet any of its ~~other~~ particular objectives more fully, since ~~As~~ that can only be done by compromising even further on the other objectives. Even more, you cannot improve a compromise by adding some new incompatible objectives which were never even thought of by the original designer. And you certainly can't do it in three months, ~~or ten months, or whatever it is~~. Pascal is a mixture of high level ideas with a number of low level restrictions which make it reasonably efficient to implement; ~~Reasonably/efficiently.~~ although it is not highly optimized. It has a lot of ~~track~~ type checking and security checking but it's not absolutely secure. There are places where it is possible to bypass the ~~track~~ type checking, but there are also good reasons why those loopholes have not been closed. It is fairly accurately defined, but not completely specified in a formal manner. ~~but the formal definitions is helpful; it has several deficiencies.~~ It is a fairly regular language, but it is not ~~octaginal~~ orthogonal in the sense of ~~our Gulf 68.~~ Algol 68. It has its irregularities and exceptions, but these are not obtrusive. And that is why the language is so easy to criticize and so difficult to improve. ~~Just a good engineering product.~~ I think you would have the same difficulty if you would try to improve the product of any of the great engineers of the past.

Why do we not set ourselves the task

~~How about setting limits on the size. First person~~ to write an operating system in less than 15,000 instructions, or less than 5,000, or less than 500, or less than 50. Why should there not be a place in the world for operating systems for each of those sizes? We don't expect all bridges to be the same length; we have different kinds of bridge for different kinds of purposes; cheap bridges and expensive bridges. And why shouldn't we design our programs ~~engineering products~~ in the same way? There is no single, marvelous, all-embracing perfect operating system anymore than there is a single all embracing perfect programming language. And the reason is really very simple: ~~perhaps I could explain it to you.~~ The main historical purpose of an operating system is to share a computer among a number of users ~~people and that's why it was started anyway.~~ Essentially, our computers at one

Now I would like to turn to the subject of engineering a product to a fixed and predetermined size, (which is surely one of the essential skills of) ~~like~~ every other engineer time ~~was~~ were so large, vast and expensive that we couldn't afford to have one of our own; so we ~~xkxxk~~ ~~got~~ banded together and bought one to share ~~and then we had to share it~~. And a shared computer is like any other shared facility. It is not quite as convenient as having our own. A bus or a train is a shared facility; it is not as convenient as having a private car. ~~Then~~ Thus the purpose of the operating system is to share a computer; but as far as possible to restore to the individual user the convenience that he had before. But of course it can't really be done. ~~It is afterall a shared facility so~~ You need ~~the~~ careful judgment about what to pretend and what not to pretend, what realities to concede ~~real~~ ~~relative to the~~ and what not to ~~and I think~~ ~~the~~ Professor Wilkes, our general chairman, in the design of the Titan operating system exercised this sort of judgment to a high degree. You don't really want in a bus to put a steering column in front of each seat. ~~The drive would be rough! And there are a number of others gadgets and few~~ However saleable, such gadgets geugaws ~~laws that~~ don't really serve any real great purpose. And once you recognize that, it's just a matter of making compromises, choosing which problems are worth solving and which ones are not ~~.~~ — a ~~A~~ matter of subtle engineering judgment. ~~and some useful things can be done~~.

~~Of course, you know~~ Unfortunately we so fall in love with our problems that when a new ~~idea comes along~~ advance arrives, like a personal (non-shared) computer, ~~which would actually make the problems go away,~~ we feel very uneasy ~~and we would like to have software on it, but when the problems back again.~~ ~~that would bring~~ because the problem of sharing has disappeared and we ask for an operating system to make it come back again.

~~Perhaps I should tell a story.~~ When I worked for computing company, ~~Elliotts,~~ I had terrible difficulty from my managing ~~director~~ because my operating system was getting too long.

Here is a little story about software size

The linkage loader ~~was~~ was nearly 300 words long. Since the previous linkage loader ~~letter~~ had been only 16 words long, it was ~~all a large order of magnitude just for which I regret to say I could give no order of magnitude justification.~~ very difficult to justify the enormous increase. The new program was certainly not twenty ~~It was not 10~~ times more convenient to use. ~~Being an automation company, the operating divisions was also wrote there in software.~~ in own software. The ~~director~~ or same manager ~~of my division, and he heard~~ shortly after a rumor that ~~the operating company~~ one of the divisions of the company ~~had written an~~ operating system of 13 instructions, and was quite impressed. After all, an operating system of 13 instructions can't be all bad. So we went to visit the division ~~had a look~~ and there they were, using the operating system. A Dozen programmers at terminals typing in machine code, ~~using this operating system.~~ ~~I'm sorry. the operating system.~~ I ~~'m~~ talked to the fellow who wrote it. He got a copy of the system out of his cabinet and he showed it to me. He showed me how it worked, ~~he explained.~~ After about 6 instructions he said "hmmm. Don't quite see what that instruction is there for. I think I'll cut it out".

Can you imagine the city fathers of London, after St. Paul's Cathedral had been built, saying "that's very fine Mr. Wren but you should ~~couldn't you~~ have added a fifth nave . That's enough about programming languages. I would like to return to our main theme.

If we cannot define a programming language that is going to serve the programmers who use it, serve their use and their convenience, then we are not likely to be able to design software products which will serve the use and convenience of their clients. It is a necessary condition of the service to our clients that our products ~~parts~~ should be well designed, accurately designed, competently implemented, and shall be precise and have the other qualities that we know about. But it is not a sufficient condition. We do wish to ~~make~~ insure that our products are easy to use. That they are inconspicious in use, we do not force our society ~~science~~ into strange and machine oriented practices in order to use our product. We have to make sure that they are designed in a manner which contributes to the goals of our society, and not ~~thought~~ to those who wish to convert them. And this is a very much more difficult aspect of engineering than the one that we have spent so much of our attention on. The establishment of sound implementation practices. And so for the next 10 years perhaps, we will see more effort expended on the design of products which are at least for their purpose and in their class models of perfection. There is no single perfect program, there is no single perfect data base system, there is no single perfect general programming language. But by restricting our objectives in a suitable way, and by placing constraints on our products, constraints of price, efficiency and ~~of~~ effectiveness, we can at least put forward the objective of designing a perfect product~~s~~. Where would music be if it were ~~was~~ the objective of every composer to compose the perfect symphony. That is impossible, but that impossibility does not mean that the composer shouldnot seek to approach our software engineering design. Maybe, in the past, there have been engineers who took this as our objectives and I like to think that maybe we could learn something about the engineering of programming languages, engineering of ~~operation~~ operating systems, editors, and other software, not by sitting down and thinking ~~up~~ all by ourselves some new product, but by looking at the product of the great engineers of the past. There have been some good operating systems. There have been operating systems which were sufficiently good ~~but~~ that they were almost not noticed by their users. Of course, these are not the perfect operating systems. Maybe they don't fit modern environment and modern machines but they were examples of perfect operating systems in their time. Of its kind the system on the ~~79~~ 7090 was a very well engineered system. It had many defects but it made a lot of very useful compromises. The TAT operating systems, which was designed and implemented by Edgar Dykstra, was a very good operating system. The operation system implemented by Maurice Wilkes and his team at Cambridge was a very very good operating system. We should study these operating systems, not only their specifications, but also the details of how they were implemented because it is in the interaction between specification and implementation that the job of the engineer lies.

I would like to suggest as a tactical research project, that we should seek, to recreate, some of the software engineering masterpieces of the past. That we should use problem structuring techniques. We should even test and develop our structuring techniques by redesigning and reimplementing, using higher level languages of notations, of some of these good products which have served their purposes in the past. Because this is how engineering progresses. Engineering is a public science. Good engineers learn their art by studying the designs of previous good engineers. Not because those designs are the perfect design but because they are each of them of their kind and in their them. Unfortunately, in programming, all these designs are encrypted in obsolete machine codes and we need a band of dedicated decrypters to ~~get~~ go in and reassess the ideas of ~~rejections of~~ these engineers in terms which we can understand — the terms of our modern use of logical notations, modern structures and the languages in this area ~~of~~ express . So for the next 10 years perhaps we can learn from not only the previous 10 years but the 10 years before that. And pay more attention to not only the internal details of the practice of our own craft but the external objectives which we are ~~expected~~ attempting to meet on behalf of our science for mankind.

Questions -

Because of the time and the nature of the Keynote address, I would like you to make your questions as specific as possible.

Question - Sir, has it already been decided or discovered or experimented or whatever, that the basic 3-structures are all that's really necessary to do anykind of programming? / the in-line, the decision and the iteration of some kind?

Answer - The answer to that question is at one level, of course, yes. At the theoretical level, it is known that those structures are sufficient and it has been known for some time. If we are an engineering science, or even a practical science we do not believe the theory until it has been proved in practice. And the theory is only useful to us insofar as it suggests experiments whereby the theory can be tested. And in this case the experiment is experiments of ~~technically and~~ taking large and difficult projects and really trying to follow through the dictates of that theory to their logical extreme. Come what may, you can not do this on a practical project, you shouldn't do it on a practical project because you can never say on a practical project which has to meet deadlines, has to meet a specification, against certain which you are going to uphold come what may, because what may come will come. But in experiment, that's the sort of experiment the universities should be engaging in now.

Question Very early in your talk you mentioned the exist~~a~~ence of the gap between the two subcultures and at the very end you gave an example which in fact magnifies it, then when you talked about the ~~P~~THE and the TITAN operating system. Now I have seen very little in the literature about ~~P~~THE system for example although the ideas and what I have read were excellent. Yet on the other hand you say they should be objects of study. Well that's where the gap is. How do people in fact study them, and in fact how do I compare the Titan system with some other one~~s~~ when I don't know how often it crashes and I don't have access to the manual so I don't even know how good it is. Why should I accept your judgment that they should be good objects of study?

Answer Well, you certainly shouldn't accept my judgment. I'm suggesting this for the future, and I realize your difficulty. Again, my suggestion is one that a great deal more work needs to be done to recreate, as it were, shall we say restore, lovingly restore the masterpieces of the past. Not because people are not painting good pictures nowadays, as well but because of their kind, painters of the past were able to produce masterpieces that are worth restoring. Until this has been done, I would not like you to take anything on trust. How it can be done, requires the same sort of loving care as is expended on the reconstruction of dead or unknown languages. It requires a great deal of devoted academic effort over a long period of time to do this reconstruction. Maybe if we can persuade our academics to do that, to recreate the past rather than always thinking they can invent something better for the future, then we will all learn from their studies.

<u>Question</u> I would like to ask where you would put languages like APL in respect to programmers tools?

<u>Answer</u> APL is a language for which I have good respect. Its implementation and design were well designed.
The language is a mixture of high level ideas like matrices~~?~~s and ~~the reyes?~~ arrays and ~~?~~ bit-maps with high level operations like multiplication of matrices and compression and so on. And these provide not only certain notations/but a sort of conceptual tool which have been used to aid in the formalization of the specification of small and large programs. And that is the sort of application or ~~instant~~ iented language that I would like to see developed. It is of course, I think, not a general purpose language and there are limitations to its applicability to large programs. For, I think, even the proponents of that language know perfectly well that any program which is longer than one line is really rather painful to write. (laughter)

<u>Question</u> My name is Noblė. I'm from Seattle, Washington where I am Director of Computer Services of the Hutchinson Cancer Center. My question is one based on my experience as a programmer which has been over a long period of time and I see the field changing. I like your analogies to master craftsmen becoming engineers because as you know it's well documented in the history of England that was a painful process for the master craftsman, particularly the blacksmith who gave up his own shop to go work in a factory learning hour discipline and that sort of thing. I see the field changing a great deal and my projections from the tutorials yesterday and from what I've learned is that from being a field that is very well paying and fairly open to new people it's going to become more restricted with - if the programmer-team model comes out where there are going to be a few queen bee analysts ~~beautiful~~ brilliant people and very many coders and I see the field stratifying some fairly

Lower levels and the field while it is expanding in numbers is reduced in status.

<u>Answer</u> We are a curious amalgam of nice papers ? One thing a software engineer can do is he can tell his client where to get off. If the client asks for something that is absurd, may be possible, but absurd, absurdly expensive or requiring absurdly ~~his~~ ~~own~~ long time scales, the engineer says, No, I'm not going to do it. If you want an engineer to expend something, if you want an electronic engineer to put a 137-channel high bandwidth rig between a ~~premium~~ magnetic tape deck and your floating ~~make a~~ point unit, he says, No, that isn't the sort of thing one should do. And his manager believe him. But if someone comes along and says, my customer or my salesman insists that you put this particular gimmick in your programming language, or put it in your product, the software man says, Yes, I think I can manage that and he draws in these great series of jumper wires which make it possible because he knows ~~heard~~ that he cannot insist; if we had more respect we would be able to resist, to make that sort of advice. The reason why we don't have the respect of course is that there ~~xx~~ in some way we haven't stratified ~~satisfied~~ our profession. That we do not recognize engineering distinction between design and implementation . Now careful, we have got it wrong before. There always used to be a distinction between a systems analyst and a programmer and it didn't work because I think it was based on an incorrect understanding of exactly how ~~xx~~ precise and logical a proper systems analysis should be. Very likely, the systems analyst should produce a formal text which is of the same order of magnitude as ~~the as~~ the eventual code it is intended to ~~be~~ implemented. We did not realize that and therefore the systems analysts themselves had a monstrous job to do and he couldn't do it properly. I would suspect that everything you say is true, that we will have people whose job is much more that of the

the inventor, the designer and that these people will outnumber us, too, very likely I'm going to be one of those people, as they do in computer design. But remember the world doesn't owe us a living. It's something that is very difficult for professors, particularly when they get to a gathering like this, to realize that the problems of the world are more serious than the problems of our profession; maybe we didn't ought to exist in such a rich and fat form (laughter).

<u>Question</u> civil engineering very much, but it seems to me the analogy is not perfect and there are two major distinctions. One of them is the precision of the interfaces required in our profession and the second is the level of complexity that we seem to become enmeshed in. Can you comment on those.

<u>Answer</u> Yes, I think you are quite right. Like all analogies, it was a compromise. I think we tend to think that we are out to prove various complex things lots more difficult than anybody has been asked to do in the past. I think it may well be true that engineering a space dropping to the moon, I would tend to regard that as a task complex and difficult as any of the spacecraft in the past. The reason why we pride ourselves, and it should not be an effort to say, pride ourselves that we are faced with tasks which are an order of magnitude more complex than any of the past, is simply because we for some reason or other are willing to undertake tasks that we don't understand.

<u>Questions</u> Joanne Miller You mentioned several areas where academia can explore for the next 10 years. Do you have any similar areas where people in industry can explore for the next 10 years.

Answer I think they probably have a difficult enough task trying to digest what we did to them in the last 10 years. I should not expect, and I do not recommend to stop? the transfer between academia and industry. In ~~our~~ other engineering disciplines the minimal interval that we would expect would be 15 years. And I think that in our fast moving profession, we are lucky if ~~of~~ new ideas ~~will not will not~~ get really transferred ~~for~~ in 15 years. In 1953, I published a little programming language idea called the tape statement - tape expressions and I noticed in 1978 that an instruction had been put into a new computer for tape instruction. That is about right. I'm happy with it.

Question I enjoyed your comment with respect to experimental processes part of the approach to ~~in~~ software engineering. But as part of that, I was curious to why you were advocating the elimination of pictures of very high level models, because that is a very necessary part of an experimental approach.

Answer Yes, I think that it would be unwise of me to deny the use of any tool to help in the incredibly difficult, I mean really impossible ~~odds~~ task of getting a grip and understanding a set of ~~inter~~ user requirements and pictures can be of the greatest use, particularly in explaining the structures and ideas ~~of some of the~~ that will convey yours designs to thei~~r~~s eventual end user. You can understand pictures if they're explained with enough words. On the whole, it does seem ~~suggesting~~ to be a fact of the nature of the material in which we work; namely, the instructions which govern the behavior of the machine. It seems to be an essentially one dimensional media. People who make maps work in two dimension. They represent the third dimension in very, almost ignore third dimension entirely. People who make cars obviously have to think in three dimensions and [illegible] because they need models. Now computers, this is logical material, logical threads that can be seen when you write programs rather wonder and if you use two-dimensional pictures to represent the one-dimensional thread, there is a danger that we will over-~~simply~~ specify or misread. Also, ~~a~~ of course ~~thought~~, it takes so very much more space, much more difficult to look up the details in the picture then it is in a well-indexed text. So use pictures for what they're worth but I suspect that the majority, the ~~good~~ great bulk of what we do ~~there~~ is in a ~~sense respect~~ linear or ~~substance~~ structured form.

Question I'd like to take issue on the question of how you design programming languages. I don't think we should be so arrogant as to assume that we are the only people who have ~~for~~ ~~them~~ this problem. I think we should learn from the lessons of other discipline take mathematics, if you have a particular application area in mathematics, what you do there is in algebra problem in the language. And although their are a great many algebras, they all have generalized frame work in common. And it seems to me thats what our program languages should provide. A generalized framework and unlike Professor Hoare, I don't like any of the languages at the present time.

Answer I like this comment. our present programming languages are We expect them to be directly to machines. ??

And that's what programming languages are now. What you are talking about I regard as even more important on use of the techniques which have already been found beneficial in mathematics in algebra to formalize an application, to formalize a specification, of a real world object, and to work when we write the computer program in order to be

we have to learn all about the things we are programming

who had ever
that I would challenge anybody/known about before.

You will have to find out more about the program

background noise, unable to transcribe

voice fading out completely