Tape green at bottom

Attach Georgia (connector)

Encouraging Confidence

But never disturb.

Tell grown of Selphine.

20mm
dumb

& through (socket plug)
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, who works in the Computer Laboratory, and works with the Program Research Group. This is another recent move for Tony Hoare, those of you who have spent known him mostly at his previous job which he held for nine years as Professor of Computer Science, King's University, Belfast. My pleasure to introduce Professor Hoare.

New speaker

This talk will be 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 defines 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 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 believes 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 we are still only after 5 months, exploring the foothills of what we hope will one day be a fully fledged engineering subject.
And so I would like not so much as to contradict our general chairman as to add to his remarks. Point out the danger that of our studies 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. They all have - all these topics are wholly program tactics, program proving, has all to do with programs, very necessary and very important. But the overall objectives that we must all seek is not some perfection in our programming methods, but rather the usefulness and the convenience of concepts but the objectives of our programming is to serve our fellow man, on which they run shall be effective to fellow beings. I would like to consider another thing which I might reconstruct as the early history of engineering.

I would suggest that our present achievements in software engineering do not stand comparison with the present status of longer established branches of engineering, whose practicalities of which it's not. Back to the days of the master craftsman. The master shipwright, the mason, the cathedral builder, the mason, the procedural builder although a remarkable experience, with highly primitive tools, and with little theoretical understanding, he somehow managed to build contract products, buildings, cathedrals, ships, chairs and tables, bridges which were sturdy, robust and even elegant and useful or spectacular. I don't know if any of you familiar with Saint Chapelle of Paris. It has fantastic walls of glass and stone faces. And how 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 started to use pencil and paper to plan his professionally drawings authoritatively and yet of course, abstract so they could determine the process for
the details of the construction of the product. He measured the distances on the line, drew and redrew the moving parts until he was sure that they would not fight against each other, that they were free to move in the right way. He then brought these designs drawn up actually by professional draftsmen, whose task it was to exercise meticulous accuracy over long periods of time. The designs were checked by checkers who spent equal amount of time making sure that they were accurate and that they all stuck together and described a product uniquely and completely. And then these designs were passed on to quantity surveyors to get an accurate and complete estimate of the 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 the production team, which people, each of whom is responsible for implementing a small part. This may take time. It may raise many problems. But if each feels responsible that his part is done properly, it is nice for when those parts are put together they will work because the design has ensured that they will. There is nothing more that can go wrong. Except, 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. He made the design inconsistent, he made them incomplete, and so I think we're quite right in our research and practical approach to engineering to concentrate on the old process of making vigorous designs of all our software products before we put them out to be implemented on a particular piece of hardware or in a particular
Language.

programming and even voluminous coding and leaving us with the task of leading starts. Then only in this way can we be sure that the product will stick together when it is put together, eventually assembled.

A product in which the management plans to spend over half of the total efforts of the project in implementing 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 knowing of the facts is which evolves from the needs of engineers to make rigorous designs of their products, have been evolved and developed and formalized in an extremely rigorous fashion and it carries the practice and tradition by which the institutes of engineering and may not be followed by every practicing craftsman and engineer. If an engineer does not follow these practices and his work fails to work, he can be sued. If his product and breaks or kills someone, there is an inquiry and he will be barred out of the profession. Perhaps engineers of the past were a little bit more fortunate than programmers, but 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, mathematical theories. We need to be able to measure, we need to be able to count, we need to be able to multiply, and a little knowledge about similar triangles and Euclidean geometry in general, how to draw ellipses, and so on, is necessary in the practice of the job of a draftsman.
15.

Now these theories are taught in quite elementary schools and they are taught in technical authorities and they are very widely practiced, and it is generally accepted that a formal understanding of the underlying principles of geometry is required of the draftsman as well as a considerable skill in practicing these skills in his drawings. I might suggest that we have found the same phenomenon in programming. Here, unfortunately, the programming practice is not pietistic. It is a logical formalism, although an austere formalism involving the more abstract part of mathematics, arithmetic, and formal logic. Of course, we can draw pictures but, approximately in general, except in such we say, instructional material, pictures are not acceptable to use for graphs. The important part is the linear logical paths and specifications. I don't think that we should expect a very logical in linear and maybe even automatic.

I don't think that we can necessarily expect to have a fully automated computerized design process when we are producing our programs. I think that if we try to bypass the phase of manual production by enlisting the aid of the computer too early, we run a severe risk of sinking under the inflexibility of our computer controlled programs, systems, whatever. We will have to involve manual practices in the same way as every other engineer would before we would determine which part of the process can be automated. I draw an analogy here it's relatively new discipline of electronic engineering and electronic system design.
A logic designer uses logic; by logic here, I mean abstract logic; logic that is studied by mathematicians. And he has a complete design of the business scene in pure logic with no physical components in it at all. And that logic design appears as it were, a complete process of the functional components of his machine. And he gets, in principal (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 the surface of a chip of silicon; in tracking wires between the components, across many levels, 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 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. And this may be something that we as programmers should not be unwilling to all. A very precise and concise 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 specifying the details of how the machine is supposed to execute the program rather than the general function which carry out. And I think that we should be willing to do this counting ourselves. I think that maybe if I can use the analogy a bit further, we have not yet discovered the discipline of logic design. The designers of our programs 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 you can conduct the whole design in terms of pin numbers and wiring diagrams of incredible length. And our programming languages that we actually use require us to specify our problems in that sort of degree of detail. How it is extremely difficult, as you can imagine, to guess the whole design of a computer correctly. 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, this programming design is so fraught with political and commercial consideration that very often when you start a project the most important decision to be made by the executive president of the whole company is that you shall program in some 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. That, I think, is only the beginning thing not to do. So in the programming languages it is one of the few branches of engineering to which I have given some thought.

Of course, since he is not working to a completed design, he has the greatest difficulty.

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 historical process. And I have seen many of them.

Earlier, with general engineering, I suggested that an appropriate analogy for the position which we are in now would be with the transition between the craftsman and the engineer. In programming languages, I'm afraid we have hardly got to the confident stage of the nearest analogy that I can think of is the alchemist. The alchemist was seeking a very worthwhile goal, so we call it the synthesis of gold. Similarly, our political masters are asking us to discover a programming language that actually makes programming easy.
18.

Obviously, it is worth spending a great deal of time and a
great deal of effort on achieving such a goal. The king who
manages to discover the philosopher'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 exact mixture of well-known ingredients
that you would have to find. 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 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 happened? After while people
learned that maybe this was a difficult thing to do; they couldn't
make gold yet, but nevertheless it was worthwhile studying the
chemical properties of matter. And so, we see a transition
between the alchemist and the chemist. And after awhile the
chemist discovered a number of scientific laws from which they
could deduce that the manufacture of gold from base metals
was impossible; physically impossible.

I suggest that maybe if we study the logic of programming
with specific care, we can see from the nature of the factor
itself that programming is relevant to the users, it is always going to be an art.

"But engineers, brilliant men, right from the point of conception down to the
detail of the implementation in octol" shall we say, on
a computer, in some code which is readable and maintainable...
19. 

"To return to the analogy with the history of chemistry in the 17th century. A little later on, of course, the physicist came along and said it was possible to produce gold by breaking up the structure of the atoms of the metal, even though it was possible, it was very expensive and was at least 150 times as expensive to make gold as to dig it out of the ground. And that economic fact is an even more impenetrable barrier than the physical one! Eventually we find that we can do it -- it can be done. But, judgement to use their automated systems..."

Where the design of program languages is obviously one of the aspects of software engineering in which good engineering principles should be employed. It is also the product of programming language, or programming languages are the basic tools of the software engineer. This is an area where understanding of software engineering, as applied to other application areas, is absolutely necessary. We must understand the programming process, both 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 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 to extend 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 if you see a good programming language is...

... 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. Hardware designers who design microcodes have an even worse, even more horrible logical tool in which to work. Yet they could use thousands of words of correct microcode, and of course scattered through are a few words of incorrect. The reason why they can do it, or would like to do it, is that the tool they are using, although it may be, it is simple enough for them to understand /in its entirety. They know in principle, 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, clear, simple, it may be complicated, but it is independent of every other instruction in the machine. The interaction effect between instructions is confined to changing one of a small number of registers; and its confined to no registered or the number of registers and the number of operations is quite small and is irregular that they can't keep control of it in their little heads. And this is why one can accomplish great feats of engineering with very simple languages. 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 manual that thick, or try it on the machine, or and he nearly always chooses 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 design, then we must have tools which we can understand and keep solely in our heads.
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, which I'll call them micro-languages, is due to the fact that such a specification must be based on an insight, a discovery of roughly equal intellectual difficulty as the discovery of the simple laws which underly the process of nature.

Without being arrogant, I hope, let us take the example of the laws of motion. I'm told that Kepler 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 earlier constructed by Tycho Brahe. He had compressed the enormous amount of random complexity into just three laws. Newton who came afterwards did not say "Oh, goody. Three laws; obviously insufficient. Let's make some more". He managed to make progress in the reverse. So I would like to suggest that in doing research on programming languages and software and this abstract software engineering, we should try to discover how small a language is adequate for our purposes. This is a matter of genuine research. We should get our research students not to cobble together some anthology of main features and then see if they can implement it before three years. But rather the reverse. That's of course a difficult challenge. Nevertheless they should throw out as much as they possibly can from his favourite language, and then show 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. Let us see what we can throw up; we know certain things are difficult. We know the chances are very difficult. We know references are looking tricky. But what would happen if we actually took our courage in our hands and threw them out. We can't do that in a practical programming project; it would be too risky. 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: who is going to be the first person who writes an operating system without a dump? The first person who writes a compiler without any references? All these pinnacles or depths are waiting to be found.
ILL, how are we going to make our languages simple. Of course, there are two ways of doing it. And I would like to see research in both directions. Once we can make our languages more specialized toward the problem areas, 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 and expressive and thoughts and maybe even the avoidance of jumps.

We can transfer that understanding to more specialized languages, specialized for example to statistical applications, special social sciences, symbol manipulation, matrix calculations. All these special areas. Ready needs of specialized languages with concepts built in which are principally for that particular application area. There I think there could be a family of such specialized languages in this area. I would like to see them somewhat notationally consistent. It would be convenient if they all used the same control structure, the same ways of defining functions, and parameters. But they should be differentiated.

In addition to eliminating features which are believed to be of doubtful safety and usefulness, how else
and 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 there is no reason why they couldn't have a reasonably standardized keyboard layout with and a reasonably standardized display.

(to simplify a language) its level of abstraction

The other thing that we can do is to specialize to level.

There are such things as high level languages and there are such things as low level languages. Sometimes we need to use one and sometimes we need to use another. Sometimes we need to use both, and make a transition between these. 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 mathematics, concepts, sequences, mapping and functions. These

low level ideas are derived from the structure, historical structure of our computers and includes things like addresses, but

jumps (machine architecture)

and a language like Pascal, which is almost as powerful as mathematics. Of course when they got 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 different languages, not only because there are all sorts of different design goals, but because there are different levels of language. A high-level language will often be sufficient for a particular implementation, but other languages might be used for implementing them, and in particular a low-level programming language should be used to write programs for implementing them. A programmer should be able to write a program in one language and then use another language to implement it. Unfortunately, in practice, such a design is very complicated, and it has a lot of subtleties that must be considered. The laws of language must be learned, and the laws of other languages must also be learned. It might well be a good idea to have separate high-level and low-level languages, but it is not clear whether this will be the best way.
Now I would like to explain why I think the levels, there is also complexity and unpredictability of modern languages arise. So as an engineering product which have been designed, I think according to engineering principles, I would like to put a small plug for PASCAL. 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 and in a fairly concrete way. Of course, it doesn't give you as much control over machine details as more machine oriented languages might. Of course it doesn't give you the same back facilities for abstraction as a really high level language like FORTRAN. But it is a compromise between these two things. Like all compromises, it can be criticized in all directions. But like all well engineered products, it is extremely difficult to improve. Every detail of the compromise has to be 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 for writing in software whereas another division had invented a language of their own which shall be nameless. This of course offends management's acceptabilities and they thought that the impact of the software company would be greatly improved if they could have both software writing divisions 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. Three months later the committee reported that there were seven votes in favor of PASCAL and seven in favor of the other language. So the management reconvened the committee and said we will give you another three months and you'll design a language of your own which will be an amalgamation of the best features of both your two languages.
Three months later they came up with a report which was, shall we say, incomplete, inconsistent and incomprehensible. They were given another three months to tidy it up a bit and then get on with the implementation. As far as I know, that was five years ago, so cut a long story short, nothing has been implemented, nothing has been decided, they are still designing.

As in deed one might have predicted. It is very difficult to improve Pascal. I would suggest that even the original author of the language himself when he moved from the original Pascal to the Mark II version, in fact made no improvement at all, no measurable improvement. The reason is that it is a compromise. You do not improve a compromise by asking it to meet any particular objectives more fully. 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, although it is not highly optimized. It has a lot of track checking and security checking but it's not absolute. There are places where it is possible to bypass the track checking, but there are also good reasons why those loopholes have not been closed. It is fairly accurately defined, not completely specified in a formal manner, but the informal definition is helpful; it has several deficiencies. It is a fairly regular language, but it is not orthogonal in the sense of Page 68. It has its irregularities, exceptions, but these are not defined.

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 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; we have different kinds of bridge for different kinds of purposes; cheap bridges and expensive bridges. And why shouldn't we design our engineering products 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 couldn't explain it to you. The purpose of an operating system is to share a computer among a number of 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 in a fixed size (which is primarily one of the essential skills of the software engineer).
time was so large, vast and expensive that we couldn't afford to have one of our own, so we bought one together and bought one to share. Then we had to share it. And a shared computer is like a shared any other 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 the purpose of the operating system is to share 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 after all a shared facility so you need the careful judgment about what to pretend and what not to pretend, relative to the

Thus, I think 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

However, we bought and there are a number of other gadgets and gizmos 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 matter of engineering judgment, and some useful things can be done. Unfortunately...

Of course, you know we so fall in love with our problems that when a new idea comes along like a personal 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, 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 Elliott, I had terrible difficulty from my manager. Manager because my operating system was getting too long.

Here is a little story about software size
Tape 2, Side A, Page 3

The linkage loader was 300 words long. Since the previous
linkage letter had been 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. It was not 10 times more convenient
to use. Being an automation company, the operating division
was also interested in software. The director of our
manager shortly after 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.

At the operating system. I'm sorry,
talked to
the operating system. I'm the fellow who wrote it. He
got a copy
out of his cabinet and he showed it to me. He showed me how
it worked, he explained. After about 6 instructions he said
"hmm, don't quite see what that instruction is there for. I think
fill out the job."
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 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 products our clients that our parts should be well designed, accurately designed, competently implemented, and shall be precise and have sufficient the other qualities that we know about. But it is not a/condition. We do wish to make sure that our products are easy to use. That they are inconspicuous in use, we do not force our 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 in their class models of perfection. There is no single perfect database 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 or effectiveness, we can at least put forward the objective of designing a perfect products! Where would music be if it was the objective of every composer to compose the perfect symphony. That is impossible, but that impossibility does not mean that the composer should not seek to understand of another musical language in which...
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 operating systems, editors, and other software, not by sitting down and thinking the 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 they were almost not noticed by their users. Of course, these are not the perfect operating systems. Maybe they don't fit modern environment but they were examples of perfect operating systems in their time. Of its kind the system on the JPL was a very well engineered system. It had many defects but it made a lot of very useful compromises. The TAT operating system which was designed and implemented by Edgar Dijkstra, 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 masterpieces of the past. That we should use problem structuring techniques. We should even test and develop our structuring techniques by re-designing and re-implementing, 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 perfect way.
them. Unfortunately, in programming, all these designs are encrypted in obsolete machine codes and we need a band of dedicated decrypters to get in and reassess the idea of rejections of these engineers in terms which we can understand—the terms of our modern use of logical notation, modern structures and the languages in the 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 the internal details of the practice of our own craft but the external objectives which we are expected to meet on behalf of our science 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 3d structures are all that's really necessary to do anykind of programming? / the 1m line, the decision and the eration 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 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 existence 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 PHE and the TITAN operating system. Now I have seen very little in the literature about PHE 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 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 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 as well good pictures nowadays, 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.
Question  I would like to ask where you would put languages like APL in respect to programmers tools?

Answer  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 and the vector and hit-maps with high level operations like multiplication of matrices and compression and so on. And these provide not only certain notations/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 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)

Question  My name is Noblé. 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 model comes out where there are 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.

Answer  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 absurd 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 my between a premium magnetic tape deck and 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 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 is in some way we haven't 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 is precise and logical a proper systems analyst's job 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 job analysts themselves had a monstrous /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
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).

Question 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 emmeshed in. Can you comment on those.

Answer 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.

Questions 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 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 new ideas will not get really transferred 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 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 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 task of getting a grip and understanding a set of the requirements and pictures can be of the greatest use, particularly in explaining the structures and ideas of some of the designs to their eventual end user. You can understand pictures if they're explained with enough words. On the whole, it does seem 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, but not a great deal.
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-simplify or misread. Also, 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 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 this problem for them. 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 that 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 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, formalize a specifica-
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