H4.SMR 282/13

SECOND SCHOOL ON ADVANCED TECHNIQUES
OF COMPUTING IN PHYSICS
(18 Jan. – 12 Feb. 1988)

"DO PHYSICISTS NEED SOFTWARE ENGINEERING?"

F. James
CERN, Geneva

10 December 1985

Do Physicists Need Software Engineering?

F. James

Data Handling Division, CERN, Geneva, Switzerland

Abstract

We attempt to describe the situation in software development which led to the invention of various techniques now known collectively as Software Engineering. This in turn leads us to consider to what extent real computer users, and in particular computational physicists, could profit from the use of these techniques.

CONTENTS

1. WHERE ARE WE AND HOW DID WE GET THERE?

What a marvelous invention is the stored-program computer! A truly universal calculating engine, it will perform any combination of logical or arithmetic operations at great speed. All the user has to do, for any particular calculation, is to tell the computer exactly what it should do, in the form of a set of instructions called a program.

It all sounds so simple, and indeed it can be, provided the problem to be solved is "small enough" to be easily understood. But then we would hardly need a computer, would we?

Here already is the basic contradiction: We expect computers to solve problems that are too complicated for our heads alone, and yet the precise instructions for solving the problems must come from those same heads. How should one set about writing a program to solve a complicated problem, and is there a way to verify that the problem is in fact being solved correctly?

These are among the questions that Software Engineering tries to answer. In this paper I examine the extent to which physicists can profit from learning and using the techniques of Software Engineering.

1.1 Early History.

When the first commercially-made computers became generally available in the late 1950's and early 1960's, the writing and operating of a program was an exercise involving intimate contact with the computer hardware and little concern for more abstract ideas of what we would now call "good programming". In those days, when computer memories were relatively small and slow, the complexity of problems which could be handled was much less than now, and the principal challenge to the programmer was to achieve maximum efficiency on the machine available. A "good program" was one that would save a few words here and some milliseconds there; any trick was acceptable if the program worked. No thought was given to the possibility that someone may have to maintain, modify, or even understand the program again at some later time.

1.1.1 FORTRAN

Even the earliest high-level languages were designed specifically to be efficient and to translate easily to the basic machine languages of the time. The most successful of these languages, FORTRAN II, was designed specifically for the early range of IBM machines (704, 709), and even though both FORTRAN and computer architectures have both changed considerably over the years, it is still true that FORTRAN is the most efficient of high-level languages for most numerical calculations.

Physicists generally appreciate FORTRAN because it is efficient, and also because they can learn it easily and because it does more or less what they want. If the average physicist were to read the computer science literature, he would be surprised to find that Dijkstra has said: [1]

> "The second major development on the software scene that I would like to mention is the birth of FORTRAN. At that time this was a project of great temerity, and the people responsible for it deserve our great admiration. It would be absolutely unfair to blame them for short-comings that only became apparent after a decade or so of extensive usage: groups with a successful look-ahead of ten years are quite rare! In retrospect we must rate FORTRAN as a successful coding technique, but with very few effective aids to conception, aids which are now so urgently needed that time has come to consider it out of date. The sooner we can forget that FORTRAN ever existed, the better, for as a vehicle for thought it is no longer adequate: it wastes our brainpower, and it is too risky and therefore too expensive to use. FORTRAN's tragic fate has been its wide acceptance, mentally chaining thousands and thousands of programmers to our past mistakes. I pray daily that more of my fellow-programmers may find the means of freeing themselves from the curse of compatibility."

We may wonder how it is that such a terrible language has nevertheless been used so successfully for so long. If he doesn't like FORTRAN, what does Dijkstra recommend that we use instead? At the time he wrote this, he in fact recommended ALGOL, although he admitted that it was also not ideal.

1.1.2 ALGOL

ALGOL was the computer scientists' answer to the shortcomings of FORTRAN. For the purposes of this paper, it is not necessary to distinguish between the various dialects of ALGOL (58, 60, 68), let us say only that ALGOL is problem-oriented rather than machine-oriented, and as such it became the "respectable" language, although people largely continued to use FORTRAN to write real programs. An absurd situation quickly developed, whereby new programs would be written and used in

FORTRAN and then translated into ALGOL if they were considered of enough interest to be published. Readers of the journal would then translate the ALGOL programs back into FORTRAN in order to use them. This situation was finally rectified when the ACM journals decided to accept programs and algorithms in FORTRAN. Nowadays many journals publish algorithms and small programs, always in their original language, and this is, for physics and related numerical analysis, usually FORTRAN.

We now know that ALGOL was essentially inadequate for writing real programs (see below for definition of a real program). Input/output was not even mentioned in the language definition,[1] good compilers did not exist, and there was little or no run-time support for such things as file handling and error recovery. But for the computer scientists, ALGOL was much more than just a programming language. It was the first of a succession of so-called "ALGOL-like" langauges, of which the most successful have been PASCAL, MODULA, and pseudo-code.[2]

1.1.3 The Credibility Gap

The conflict between advocates of FORTRAN and ALGOL was both a cause and a symptom of a situation of mutual mistrust and scorn between physicists and computer scientists which we may call the Credibility Gap.

On the one hand, physicists were interested in producing running programs to solve often complex physics problems, and it was clear to them that computer scientists (whom they suspected of never having actually written a program) had nothing to offer but bad advice.

On the other hand the computer scientists, armed with mathematical proofs of the superiority of their methods, had only scorn for the masses of applications programmers who were turning out thousands of lines of inelegant code filled with undiscoverable bugs and inaccuracies.

---

[1] Of course ALGOL programs perform input/output, but it is not "standard". In fact, no true international standard for ALGOL 60 was ever made, whereas the first FORTRAN standard appeared in 1966.

[2] Pseudo-code is not intended to be compiled, but is heavily used for precise descriptions of algorithms.

As a result of the Credibility Gap, computer scientists benefitted very little from the experience of those working on big scientific programs, and physicists tended to ignore the more elegant methods, tools, and algorithms produced by the computer professionals. Fortunately, this situation is now improving and the two worlds are moving closer together, but much progress can still be made. One area where physicists are beginning to recognize the possible value of computer science is that of Software Engineering.

1.2 Case Studies.

Software Engineering was born out of the realization that software was becoming a major bottleneck in many projects, that software was consistently delivered later than expected and full of bugs or not working at all.

In order to overcome this problem, it is necessary to understand how it arises, and for that we need information about just what happens to individual projects. Unfortunately, detailed case studies are rare, especially for unsuccessful projects. After all, who wants to make a detailed report on his project after it has failed? Even if it has worked, the inevitable pressure to improve it or to embark on a new project is usually overwhelming, and in any case, people working on real projects are paid to make them work, not to describe their success or failure.

Fortunately, there are a few organizations so big that they can afford to invest some of their time in seeing what they do the rest of the time, and this has proved to be a sound investment for the entire data processing community. One such organization is IBM, and they have produced innumerable software projects, of which two have become especially famous: OS/360 and The New York Times Information Bank.

1.2.1 IBM OS/360

In 1964, Frederick Brooks became project manager for the writing of OS/360, one of the first real computer operating systems, and probably among those that will live the longest. This enormous project, believed to have cost about $200,000,000 in 1973, proved to be far more complex than originally expected. Although it is not explicitly the subject of a case study, it has inspired several works by Prof. Brooks, of which the most famous is 'The Mythical Man-Month'. [2]

On balance, one should probably consider the OS/360 project to have been a success, since it did ultimately get written and has been used by untold thousands of not-always-happy customers, but the most successful part of the project may turn out to be the writings of Brooks, who has analyzed with clarity and some humour, the pitfalls of such a project. The meaning of the title 'The Mythical Man-Month' is that it is NOT true, at least for a software project, that you can multiply the number of men times the number of months to calculate the total software output. ("This may be true of picking cotton", he says.) We discuss some of Brooks' results below.

To those of us who have had to suffer under the burden of using IBM JCL, it is very enlightening to learn about the OS/360 project, since it then becomes clear that it was not at all intended that IBM operating systems should look like JCL. That was simply as far as things had got when the project ran completely out of resources (time and money).

1.2.2 The New York Times Project.

In contrast to the OS/360 project which was begun in the pre-Software Engineering era and serves as a model of what can go wrong, the project (also undertaken by IBM) to write an information retrieval system for the New York Times is often taken as the proof that Software Engineering really works. Since it is considered as a great success, it is not surprising that it is probably the best documented case study of any major software project.

The new idea used in the NYT project was the "chief programmer team", and those responsible are clearly proud of the whole project. More recent criticism of the project indicates however that the "chief programmer team" concept is no longer considered positively, and it is no longer so clear that the project was even very successful in terms of the difficulty later encountered in maintaining the system. A lesson to be learned here is that case studies, especially when written by the people involved directly, may be extremely optimistic about the quality of the final product and tend to ignore important problem areas.

1.2.3 More Recent Studies.

Since around 1980, publication of case studies has become more common, although they mostly suffer from the same defects as before. The U.S. Department of Defense, for example (one of the few institutions bigger than IBM), has produced quite a few.

In addition, a further phenomenon, the "Survey" or "Statistical Study" has proved exceptionally useful. In a Survey, a computer science research worker interviews many people responsible for software projects, collecting and publishing the results statistically, without referring to any project explicitly. This has several advantages:

- The project leaders are more likely to be honest about the success or failure of their project since it is not identified in the sample.
- The projects to be sampled are chosen by the research worker, who is not involved in the projects and therefore chooses them according to some objective criterion not connected with the success or failure of the project.
- The results allow easily to distinguish between typical and exceptional properties of a project.

1.3 The Life-Cycle

One of the major advances in Software Engineering which grew out of the case studies has been the realisation that a major software project has its own "life", generally characterized by certain rather well-defined stages which make up what is known as the life-cycle. The study of the typical life-cycle does not directly help us to write better programs, but it certainly does help in planning the resources that will be required, and in avoiding the nasty surprise which may come when we realize that our project, after all, is not behaving any differently from all the others.

The various stages through which a large programming project generally goes during its existence may be characterized in different ways, but the following steps are usually identifiable:

1. *Feasibility and Requirements Analysis:* Deciding what the program should accomplish, and whether it can be done.

2. *Logical Design:* How the program should go about attaining its goals. The exact flow of data, logic, and control.

3. *Detailed Design:* How the logical design should map onto the computer system (subroutine structure, data structure, etc.).

4. *Coding:* Actually producing the code (for example, Fortran) of the program.

5. *Implementation:* Installation on the chosen computer system(s), integration with appropriate libraries or other tools, provision of user interfaces, documentation, etc.

6. *Maintenance:* Insuring that the program continues to meet the user's needs, which involves:

- Fixing bugs.
- Installing program on new computers.
- Modifying program to adapt to changing environment (for example, changes in operating systems or file formats).
- Modifying program to adapt to changing user requirements.
- Improving program and adding new features.
- Preparing conversion to new program when this one finally becomes obsolete, if ever.

A naive approach consists in considering only step 4 (coding), since that is the only one which must necessarily produce a tangible output, namely the program, and once the program is written one might think the project is over. Studies of actual projects show however that steps 5 and 6 invariably consume at least as much programming effort as step 4, and for large projects can take up many times the effort and over a much longer time span.

Steps 1 to 3 (requirements analysis and design) may not even be explicitly recognized, especially in a small project, but one of the main themes of Software Engineering is the importance of investing sufficient effort in these steps before beginning to write code. It is believed that for large projects the total effort is minimized when more effort is spent on design than on coding.

1.4 The Software Crisis.

As computer hardware performance has improved, it has slowly become clear that software performance is not able to keep up. Programmers who were somehow able to squeeze results out of the earliest machines are simply no longer capable of producing good enough software fast enough to take advantage of the possibilities of the newer generation of computers. With hindsight, it should have been obvious that the capabilities of the human brain could not be expanded by orders of magnitude as has hardware performance, but no one seems to have realized what was happening until it was too late. The Software Crisis has arrived.

Although theoretically it is still true that anything can be done in software, it has became painfully obvious that it is simply not going to happen. Software also requires resources, especially time to write it, and it will not just appear miraculously when needed. Software projects are consistently late, far more expensive than foreseen, and usually give disappointing results. Programs that apparently work still have to be improved, modified, maintained, transported, and constantly debugged, an effort which usually turns out to be far more time-consuming than the original writing of the program. Even the word "project" is probably a misnomer, since it implies a termination, whereas real software projects seem to continue endlessly, to a point where most programmers are spending most of their time maintaining old programs and producing almost no new ones.

One obvious way out of the Software Crisis is to purchase software rather than developing it yourself. This method is being adopted by more and more computer users as they realize that developing software does cost real money, even if they have people around who are good at programming. Software budgets then tend to rise as hardware budgets remain stable or decrease. It was not long ago that computer manufacturers supplied all the basic software free with the purchase of a new computer, whereas today the software and documentation necessary to operate reasonably a new computer may cost as much or more than the hardware, and it has even been predicted that in the not-too-distant future software houses may supply free hardware to run their products!

1.5 The Theory of the Software Crisis.

Just what is it about computer programs that causes them to have bugs? Why do they defy logical analysis? Are they fundamentally all that complicated?

A simple example shows that they are indeed complicated. Consider a subroutine containing just one DO-loop which is traversed ten times, and which contains logical IFs such that there are six different possible paths through each traversal of the loop. Then the total number of different possible paths through the subroutine is 6^{10}, about the number of seconds in two years. If the subroutine contains two such (independent) loops, the total number of paths is not doubled but squared, and is approximately equal to the number of microseconds in a century. Clearly it will be impossible to analyze the behaviour of such a subroutine by explicit consideration of all possible paths through it.

In a recent and already famous paper, Parnas [3] has pointed out that it is precisely this discrete nature of programs that makes it impossible to analyze them in the way traditional engineering problems are analyzed. Other engineering disciplines are based on continuous mathematics, where a small change in the input causes a correspondingly small change in the output; in digital computing however, we all know too well that changing a single bit in megabits of memory is usually enough to make a program fall on its face. The discrete mathematics (logic) required to analyze such systems is very complicated and less familiar to most of us. In fact, the only complex discrete systems which we can really understand currently are those involving a high degree of repetitivity (such as the hardware chips), and real programs generally fall outside that domain.

The inevitable conclusion is that big software packages will continue to be unreliable unless we can either:

- learn to write them in such a way that they can be analyzed by the meager mathematical methods we have available for discrete systems, or
- develop a more powerful mathematical logic analysis tool, or
- find out how to write fault-tolerant software.

We don't currently know how to do any of the above, but future research in Software Engineering may conceivably lead to advances in these areas.

1.6 Physicists and the Software Crisis.

Do physicists know about the Software Crisis? In principle, yes, they can tell you that it exists. Most are able to repeat mechanically the well-known phrase: "Hardware is getting cheaper while software is becoming more expensive". But many of them still behave as if software came for free.

At CERN, for example, there is currently considerable debate about the purchase or development of software packages in such areas as data base management, code management, graphics, FORTRAN verification, SASD tools (see below), networking, and others. Those proposing in-house development still talk in terms of "man-years to finish the project", as though men and years were interchangeable, and as though a project would consume no further resources once it was completed. And this in spite of the fact that in some cases the person proposing the project is already spending a large part of his time maintaining projects he "completed" years ago.

Another symptom showing that physicists believe software to be free, is that it almost never appears explicitly as an item in the budgets for big experiments, even though these experiments have budgets of tens of millions of dollars and will depend critically on a very large battery of computer software, most of which they will produce themselves.

There are however signs of change in the air. The sudden explosion of interest in Software Engineering among physicists, even though still spread very unevenly, shows that many of us are sufficiently aware of the Software Crisis to investigate ways of alleviating it. The big question is whether Software Engineering can really get us out of this mess.

2. REAL PROGRAMS.

In order to clarify just what the Software Crisis is, it is good to consider what real programs are expected to do. This is largely for the benefit of theoretical computer scientists who may never have been confronted with the ugliness of a real program, since we may need their help in coming to grips with the complexities of such programs. For the purposes of this paper, we may define a real program as one which possesses all or most of the following properties:

1. It requires at least a few thousand lines of high-level code for a slave utility package, and many thousands of lines for a master project.

2. It will consume considerable amounts of computer resources (cpu time, real time, memory, disc space, etc.) in its lifetime, so efficiency is a major concern.
3. It will be used on several different kinds of computers, including at least one which was not considered when the project was started.
4. More than one person will participate in designing, writing, maintaining, and documenting the program.
5. Many people will use the program, including people with different degrees of expertise in its field of application.
6. The problem to be solved is sufficiently complex that no detailed specifications exist. If by some miracle there are detailed specifications, they will change considerably during the project or else they will not correspond to what the end users really want.
7. No matter how general the program is, someone will have to modify it for his purposes.
8. It may or may not perform floating-point operations, but it will do input/output operations of several different kinds (e.g., formatted, binary, random-access, interactive, graphical, network, etc.) and also require some other interface with the operating system, for example for error recovery.

2.1 User Interface.

The user interface is often a key part of a real program, since the program will most likely be operated by and for the benefit of people (users) and must therefore be instructed as to how it should proceed with a given task for a given user. The best form of interface will depend very much on the context and application, and will be different if the program is, for example, a slave utility package or a whole system which is itself driving lower-level modules.

Often the interface will take the form of an interactive dialogue with the user, allowing him to supply input in a variety of ways depending on the kind and amount required. A good interface checks the input and output data for consistency, and allows the user to verify his own data by providing him with, for example, some graphical representation. Features of this kind help to avoid mistakes and useless calculations, and reduce the user's learning time by helping him through prompts, intelligent defaults, etc.

User documentation should be considered an integral part of the user interface, since poor documentation will provoke user mistakes just as an error-prone software interface does. An undocumented program may as well not exist, for it cannot be used, and good documentation is even rarer than good programs.

2.2 Portability.

There are real programs (operating systems, for example) which are written for one particular type of hardware and never run on another, but in most application areas it must be considered a severe restriction on a program. In my experience, if a program is really useful, someone will eventually want to install it on a computer for which it was not intended, and it is better to plan on that in advance. It is of course tempting, when the target machine is known, to exploit all the convenient peculiarities offered on that machine, even if they are not portable, but one usually ends up paying double for this laziness when it comes time to convert to another machine.

2.3 Maintainability.

Portability is really just one aspect of the wider problem of maintainability. It would be nice if a program could simply be written, installed, and abandoned to the users, but that is not how real programs work. Not only do computers and operating systems change during the useful lifetimes of real programs, but users' requirements change, bugs are found, improvements are requested, and none of these changes can be foreseen exactly. All this means that programs have to be maintained, and it is no secret that some programs are easier to maintain than others. If a program's logic is so complicated that even the author cannot remember how it was supposed to work, it may become impossible to maintain and eventually have to be abandoned even though it originally performed exactly as required.

2.4 Efficiency.

Toy programs can sacrifice efficiency for higher virtues, but real programs cannot. Even with computer memories becoming ever cheaper and bigger, we are still forced in many cases to resort to unpleasant techniques such as overlaying, memory management with garbage collection, and temporary storage on disc files in order to fit our programs and data into the available space. Similarly, the ever more powerful central processors still do not exempt us from the bother of optimizing cpu efficiency for programs that are going to consume thousands of hours of computer time.

2.5 Input/Output.

Although many computer scientists would prefer to forget about I/O (some even propose that we use programming languages in which I/O is not defined!), real programmers cannot. Real programs need input and output, and it often poses problems. The user interface is normally implemented in terms of formatted I/O, often interactive and/or graphical. The basic data input and output may involve megabyte rates, so it has to be efficient as well as portable (if possible). And I/O exception handling, even though it should be invoked only rarely, is often a most important aspect, and may account for a large part of the code of a real program.

3. HOW SHOULD WE WRITE REAL PROGRAMS?

3.1 Program Project Management.

If a programming project is big enough, it will consume a considerable amount of resources: manpower, computing facilities, and other resources, especially time. These resources must be managed. The people working on the project want to know what they are supposed to do and what they will have to work with, and those paying for the project want to know how their money is being used. This means that even in a physics software project (which tends to be more informal than a commercial or military one) there must be at least one person to take care of the following management tasks:

- Predicting how the project should evolve in time and making sure that actual progress is sufficient.
- Making sure that the different people participating in the project are doing what they are supposed to, that they are cooperating and communicating with each other. Much time is wasted when one person refuses to use a subroutine written by someone else, or insists on rewriting perfectly good code.
- Seeing to it that everyone has proper facilities for working.

These very important spects of software development are treated extensively in the literature. One very good book is that of Boehm [4] .

3.2 Program Design.

The Software Crisis had taught us that programming was a subject to be treated with great respect, and various theoretical exercises and practical experience had taught us something about good and bad ways to program, but programming had not yet really become a methodological discipline like a branch of engineering. It was more like an art or a craft. Something important was missing, and people looked to the methods of engineering for inspiration.

After all, what does an engineer do when he wants to build a bridge? He certainly doesn't start by ordering the steel and digging holes. He starts by looking at the **specifications** for the bridge, such as required width and load to be supported. He then **designs** the bridge, making drawings to determine dimensions, and calculating the required amounts of steel and other materials. He can then estimate how much it will cost and how long it will take to build. He can show his plans to non-technical people such as politicians and bankers to get their support for the project and perhaps modify the project to take account of various constraints or changes in specifications. No one would consider starting to build the bridge until all this preliminary work was done.

Can we and should we proceed in an analogous way when writing a program? Perhaps we should not just sit down and start coding, but rather put more time into the preliminary stages of specification and design. But how can you specify and design a program? How can you draw a picture of it, calculate how much it will cost, how long it will take to write, and how big it will be? How can you show a plan of your program to a politician or a banker, or even a physicist or a laboratory director? How can you modify your design in the light of comments by your director in case he finds it too expensive or not good enough? And how can you do all this before writing a line of code? Is it even a good thing to attempt to proceed in this way?

All these questions have led to the development of Software Engineering, and in particular its highest level, Software Design Methodologies, an attempt to make software development into a proper engineering discipline. The goal is certainly noble, and there is little doubt that progress has been made, but you will be very disappointed if you expect Software Engineering to look anything like Electrical or Mechanical Engineering.

3.2.1 Boxology.

The heart of any program design methodology is the part that allows you to make diagrams that represent the program to be written, much as an engineer would make a drawing of a projected bridge. Since programs are not tidy three-dimensional objects, it is not easy to represent them on paper in an abstract way. Attempts to make such a representation generally end up with boxes, bubbles, or

lozenges of various shapes connected by lines of different kinds, so the study of these representations has been called "boxology".

A good introduction to boxology is the tutorial of Freeman and Wasserman [5] . Thumbing through the 455 pages of this tutorial quickly gives the impression that the poor physicist trying to find his way through the jungle of boxology may be little better off than the prehistoric beasts struggling in the tarpits as evoked by Brooks [2] .

Indeed the danger of spending a long time choosing and finally learning a design technique which may not turn out to be worthwhile, is very real and has discouraged more than one well-intentioned project manager. Still some of these methods have been used successfully in real projects. Without trying to single out one method as better than the others for all applications, it is good to mention at least one program design methodology which has been used even in the high-energy physics community, namely SASD.

3.2.2 Structured Analysis and Structured Design.

The basic principles of SASD are presented in a book [6] of M. Page-Jones. Of the various techniques which make up SASD, we discuss only its boxology.

SASD diagrams are probably not as clear or precise as engineering drawings, but experience shows they really do help the design team to know what it wants to do, to discuss the project within the team, and — very important for the project boss — to explain to those not directly involved in the program development precisely how it intends to write the program. There are in fact two different flavours of diagrams in SASD:

- **Data-flow diagrams,** used in conjunction with a **data dictionary** represent the functional design of the program by indicating the flow of data between different logical processor modules. A given diagram should show only five to ten different modules. Each module shown on the highest level diagram can then be broken down into sub-modules in a lower-level diagram, and this can be continued to any desired depth, in such a way that each diagram indicates a more detailed design of a module appearing in a higher diagram, with the top-level diagram showing the overall program design. Similarly, the data dictionary defines each data entity in the data-flow diagram as a set of data items, each of which may in turn be defined as a set of sub-items, etc. In this way, a human being reading the diagram or the dictionary can understand the whole by understanding separately each part, never having to consider at one time more than a small number (five to ten) of processes or data items.
- **Structure charts** show the program design in terms of computer-oriented entities like subroutines and files. Of more use to the programmer than to the project boss, these charts indicate how the functional design should map onto the computer itself, but still without explicitly invoking the programming language which will be used to write the programs. If well written, structure charts allow the end programs to be written by programmers with little or no knowledge of the actual field of application of the programs.

3.2.3 SASD Tools

At this moment, the greatest weakness in the SASD methodology as seen by the physicist user, is the lack of adequate computer-based tools to help him to create, modify, and present the various diagrams and dictionaries. It turns out in practice that the diagrams evolve continuously, requiring many trial versions before converging to the finally accepted ones. There is nothing "wrong" with this; it is symptomatic of the fact that SASD is indeed helping the user to see exactly what he is designing and to suggest where the design is weak and must be modified. This means however that the user needs good tools to help him modify and display the diagrams. Such tools do exist, but all those currently available suffer from one or more of the following inadequacies:

- Too expensive or not generally available.
- Poorly documented or no HELP facility.
- Not portable to different systems and output devices.
- No verification of consistency between diagrams of different levels.
- No verification of consistency between diagrams and dictionaries.
- No facilities for interactively modifying and redisplaying diagrams.

Because of the importance of these tools, there is considerable activity in this area. Some useful products do exist now, and it is likely that really good tools will be available soon.

3.2.4 Physicists and SASD.

One of the most extraordinary and unexpected (at least by me!) phenomena in the recent history of computational physics is the sudden interest in SASD on the part of experienced physicist programmers. Many of these people, who have the reputation of being brilliant but hard-headed and very resistant to change, are now realizing that Software Engineering may have something to offer, and are even willing to invest considerable time in learning and applying the techniques of SASD. Even some of the most inveterate bit-twiddlers, the kind that revels in machine language and microcode and normally can't be bothered with any kind of documentation, can now be seen producing data dictionaries and discussing their projects in terms of data flow diagrams.

Of course the revolution has not yet converted everyone, and many are bound to be disappointed by some aspects of SASD, but at least it has become a respectable thing to know, even to the point that those who are not yet initiated may appear somewhat ashamed of their ignorance.

3.3 Programming Techniques.

This section is concerned with the way programs are written on a microscopic level.

3.3.1 Program Complexity and Proving Correctness.

As indicated earlier, it turns out to be far from an easy business to prove even a relatively simple program correct. Such a proof for a non-trivial subroutine is given by Hoare [7]
, but one does not find many others, in spite of the amount of effort which has been expended in this area of computer science. Those who hope to find clear mathematical proofs of the kind we might find in a textbook on geometry or linear algebra will be very disappointed in computer program proofs.

However, the effort put into trying to prove programs correct has paid off handsomely in a perhaps unexpected way. As people found they could not prove complicated programs easily, they began to consider restricted classes of programs constrained to follow certain simplifying assumptions. Certain of these constraints made program proving enormously easier, and it wasn't long before people got the idea that if a program was easy to prove correct, it might also be easy to maintain, modify, and improve, as well as being surely correct. Inversely, a program not obeying these restrictions, since it could not easily be proved correct, might actually be wrong, and would certainly be hard to understand and modify. Although these ideas were not as mathematically rigorous as they had been intended, they led to the discovery of some rules for good programming which eventually became known as Structured Programming.

3.3.2 Structured Programming.

Structured Programming refers to a set of rules and techniques of actual code writing which makes the logic or structure of the program more transparent. The most important and famous rule is to avoid unconditional jumps (GOTOs), especially backward jumps. Theoreticians trying to prove programs correct found that if a program contained a backward jump, then it was often impossible even to prove that the program would terminate, let alone give a correct answer. On the practical side, programmers

debugging routines caught in infinite loops soon learned to be suspicious of backward jumps as well. Moreover, a rather theoretical study had showed [8] that GOTOs were not necessary in a language which provided adequate control structures [FORTRAN IV did not provide those structures, but FORTRAN 77 with the IF−THEN−ELSE does.]. There followed a general consensus that the backward GOTO, and in most cases forward ones as well, were symptoms of a poor, or unstructured program. A well-structured program used the control structures of the language (DO, IF, GO TO) in as clear and linear way as possible, so that it was always clear, at any point in the program, how you got there.

But good programming is more than just good control structures; there are many other aspects to consider. Anyone can recognize a poor program, but it is not so easy to describe what makes a good program, since everybody has his own ideas on the subject. Fortunately, some people have tried to study this problem as objectively as possible, and have come up with some criteria for what makes a program "good", where "good" can be defined as:

- Easy to understand
- Likely to be bug-free
- Easy to modify
- Portable to other systems
- Efficient

The particular criteria will of course depend very much on the programming language used, since some are better than others at "forcing" the programmer to write good programs. In this respect FORTRAN is relatively liberal, allowing the programmer considerable freedom to write rubbish. For this reason, the recent book of Metcalf [9] is particularly welcome. It contains a wealth a carefully considered recommendations which, if followed, will greatly improve the quality of a FORTRAN 77 program.

3.3.3 Physicists and Structured Programming.

Nearly as impressive as the revolution in SASD is the gradual interest of physicists in structured programming. Whereas the traditional physicist considered any kind of FORTRAN conventions or recommendations to be an infringement on his intellectual freedom and an unpardonable interference in the accomplishment of his work, I can now detect a general desire to write good, clear, standard FORTRAN.

The motivation behind this increased interest in good programming may have to do with the increase in software sharing and networking which makes one's own programs much more visible to other programmers. It certainly has to do with the fact that we simply can no longer afford to write such bad programs as we did in the past. In modern computational physics, the consequences of a software malfunction are just too great.

4. CONCLUSIONS.

So do physicists need Software Engineering? Yes, certainly if they are envisaging a software project of any reasonable size. The Software Crisis is here and is real. Traditional methods will simply not allow us to produce software of the quality and quantity needed.

The real problem is not whether we need Software Engineering, but rather: Does Software Engineering, as it exists today, offer us what we need to overcome the Software Crisis? The answer to this question is much less clear. Progress in most areas has not lived up to our great expectations. There are still no sure-fire techniques for writing reliable programs or meeting software deadlines. The typical physicist programmer who wants to do research in physics, not in computer science, is still faced with a doubtful trade-off between input effort and output results.

In an effort to guide the physicist through the jungle of literature and methods, I will apologize in advance for being very incomplete and venture to suggest just three works likely to improve the chances of success of a software project:

- For project managers, the book of Boehm [4] .
- For those participating in software design, the SASD methodology, and the book of Page-Jones [6] .
- For good coding principles in FORTRAN 77, Metcalf [9] .

It is my sincere belief that this list will become longer in the coming years as computer scientists come up with more and more ideas which are not just promising, but effective.

REFERENCES

[1] E. Dijkstra, The Humble Programmer, Comm. ACM, 15 (1972) 859–866. This article is also reprinted in [10]

[2] F.P. Brooks, Jr., The Mythical Man-Month, Addison-Wesley (1975). Extracts of this work have appeared in Datamation of December 1974, and are reprinted in [5]

[3] D.L. Parnas, Software Aspects of Strategic Defense Systems, American Scientist, 73 (1985), p. 432.

[4] Barry W. Boehm, Software Engineering Economics, Prentice-Hall (1981).

[5] P. Freeman and A.I. Wasserman (eds.), Tutorial on Software Design Techniques, Third Edition, IEEE Computer Society, New York (1980).

[6] M. Page-Jones, The Practical Guide to Structured Systems Design, Yourdon Press, 1980.

[7] C.A.R. Hoare, Proof of a Program: FIND, Comm. ACM 14 (1971) 39–45 This article is also reprinted in [10]

[8] Bohm and Jacopini, Flow diagrams, Turing machines, and languages with only two formation rules, Comm. ACM 9 (1966) 5. A summary of this paper by D.C. Cooper is published in [11]

[9] Michael Metcalf, Effective FORTRAN 77, Oxford University Press (1985).

[10] E.N. Yourdon (ed.), Classics in Software Engineering, Yourdon Press (1979).

[11] E.N. Yourdon (ed.), Writings of the Revolution, Selected Readings on Software Engineering, Yourdon Press (1982).