SLAC–PUB–5293
June 1990
(E)

SOFTWARE MANAGEMENT ISSUES*

Paul F. Kunz

Stanford Linear Accelerator Center
Stanford University
Stanford California, 94309
<BITNET: PFKEB@SLACVM>

ABSTRACT

The difficulty of managing the software in large HEP collaborations appears to becoming progressively worst with each new generation of detector. If one were to extrapolate to the SSC, it will become a major problem. This paper explores the possible causes of the difficulty and makes suggestions on what corrective actions should be taken.

Invited paper presented at Computing in High Energy Physics, Oxford, England, April 10–14, 1989 amd published in Computer Physics Communications 57 (1989) 191-197.

* Work supported by the Department of Energy, contract DE-AC03-76F00515.

1.0 SOFTWARE ISSUES

Today, in high energy physics, software is generally in a mess. That is to say, most experimental groups, especially the new large detector groups, are having a difficult time developing and managing their software. As each new large detector comes on line, the management the software effort seems to be getting more difficult. This seems to lead to a conclusion that for Superconducting Super Collider (SSC) we will have a major software problem. Although not explicitly stated, there seems to be many in our community that believe the reason that software will be a problem at the SSC is that "we need to develop large (200-500K lines of FORTRAN) complex code for the detector with 400 physicists at 50 institutions."[1]. We will first explore if the above reasoning is fact or fiction.

First of all, lets look at the size and complexity of the code for a very large detector. The size and complexity of the code should scale with some aspects of the detector. If we can find the scaling laws, we should be able to estimate the size of the problem for an SSC detector by extrapolation from our current detectors.

The size and complexity of the code should scale, for example, with number of different kinds of detector elements in the detector. This is because each detector type will need its own pattern recognition code and there will be some code that links tracks between the detector elements. For an SSC detector, however, there is no reason that there be more different kinds of detector elements than a large Tevatron, LEP, or SLC detector. Therefore, this scaling law would say that the software for an SSC detector would not be more difficult.

Another scaling law is the size and complexity of the code scales with the number of boundaries in the detector. This is because each irregular boundary takes additional code to calculate the position of the boundaries, cross the boundary, and in general, makes for a lot of exceptional case handling in the code. There is no reason that an SSC detector should have more boundaries than existing large detectors, so the software problem for an SSC detector should not be more complex because of this scaling law.

The size and complexity of the code should scale with the track density due to confusion a pattern recognition program must try to resolve. However, these problems are in a limited area of the detector code and the effect is not very strong. Thus we would not expect a great deal of size and complexity from this effect alone.

So far, we have seen areas where an SSC detector is not necessarily very different from our present day detectors. However, there is still a feeling shared by many that the large physical size of an SSC detector is going to lead to a larger software code problem. For example, an SSC detector will have many more detector channels. But the size and complexity of the code should not scale with the number of channels; not to first order. Only the size of the arrays should grow, not the size of the code. The same could be said about the number of tracks in the detector, except for the second order effect that with a large number of tracks one expects to have areas of higher track density. Nor should the change in energy scale of the particles in the detector have a strong effect on the code. And certainly, the total amount of iron in the detector doesn't have effect on the size and complexity of the code.

There are other effects that I would consider second order effects. With each new detector, for example, one frequently tries to push the state of the art in detector resolution, or two particle separation, or both. This will lead to more code and perhaps somewhat complex code. However, since this code should be localized in the module that does pattern recognition for the subdetector, I consider it a second order effect. Also the large number of channels will require some form of database management system. for calibration and alignment constants. But again, if well planned and modular, it should not lead to overall impression of large and complex code.

Thus we see that because an SSC detector is very large compared to our current detectors, there is no inherent reason that the code for the detector be any larger. The notion that a large detector leads to large software problems is thus mostly fiction. The second part of the reason is the people factor, which we will now explore a bit further.

Over 400 physicists are expected to be collaborating on a large SSC detector. Getting so many people working on a software project is clearly a problem. But in our modern era, it seems that only about 10% of them actually work on the Monte Carlo or event reconstruction code for a significant fraction of their time. This means the real size of the software team is about 40 people; a much more manageable number. Of these 40, we might expect them split up amongst 4 to 6 detector types. That is, for example, the ver-

tex detector, central drift detector, particle id device (if one exists), calorimetry detector, and muon chambers. If equally divided, there would be 7-10 software people per detector type. Another small team would also be working on system architecture, utilities and tools. People in industry, experienced with managing large software projects, tell us that this is about the right size for a software team[2]. In fact, today's large detectors are usually under-manned in its software development effort, leading to software teams which are even smaller. Thus there should not be an increasing software development problem because of too many people, unless of course, all the people try to get heavily involved in one area, such as the utilities and tools.

The number of institutions involved with a large detector is frequently said to be the cause of the software problem. But computer networking, as we have it today, should have largely solved the problem of the geographic dispersion of the software team. It is true that people working at remote locations can not attend as many meetings as those a the central site, but too many meetings break one's concentration on developing code anyway. There is, of course, a need for a system to distribute program libraries to remote computers, and the possibility of having the program run on multiple computer types. But the difficulty of such a distribution system should be a second order effect compared to the software problems we are experiencing. It is also frequently the case that even at the central site, one needs to run the code on at least two different machines.

To the question: "will software be even more difficult at the SSC?", I think the answer is "no, it shouldn't be". It shouldn't be because we have not identified any first order scaling laws from which we could conclude that software will be more difficult. I would argue that the reasons that are usually given for predicting a major software problem for SSC detectors is largely fiction.

And yet we know it is a fact that with each generation of large detectors, the software problem is growing. The discrepancy between what we have concluded is fiction and the fact that software is an increasing problem lies in correctly identifying the causes of our software problems. Without correctly identifying the real causes, it will be difficult to find real solutions.

2.0 Causes of our Software Problems

I do not profess to understand all the causes to the software problem. Nevertheless, I am willing to hazard some reasoned guesses. Amongst these, I do not pretend to understand the relative importance between them. Amongst the large collaborations, in fact, their relative importance may be very different.

The first cause is that some people don't take software seriously enough, early enough. The second is that some people take software too seriously, too soon. Lets explore the first a bit. Some people are too busy prototyping, building, testing, and installing the hardware to be "bothered" with software. They may think that they can always fix the software later. It is likely that some of these people are the real experts on how the detector works, thus the ones that will ultimately write the code that works. And yet, in the early stages of building the detector, how much time do these experts spend in monitoring or controlling what software is being written by others. And who are these others? They are probably younger, less experienced physicists who could very well use some guidance from those who have been through this kind of software development before. This kind of situation can lead to large important parts of the code being re-written at the last minute perhaps, even perhaps, after the first data has been taken.

Another cause is that some people take software too seriously too soon. A telltale clue that this is happening is when religious wars break out. The topic of these wars are such things as whether to use FORTRAN or another programming language, which is the "best" operating system to use, or which code or data management system should be used. An excessive amount of time can be spent discussing these issues, attempting to make decisions, and/or designing new tools. It is as if people believe that decisions can be made at this early stage of software development that are really going to last for five or more years. Hardware people know they can not make final decisions on design or construction without making a prototype first. But for software people, the word "prototyping" isn't used very often.

Frequently, when software is taken too seriously, a group builds an overly complex software architecture which is full of gadgets. These architectures frequently lack an engineering trade-off analysis, in that concerns about details that may occur 5% of the time seem to outweigh 95% probable usage. There may be a lose of contact with reality and

the end-user for whom the system is presumably designed. There may be also a lack of cost-effectiveness analysis of the gadgets, where the real costs of a gadget include not only the writing the code to support it, but also the integration, maintenance, documenting and training of users. The documentation cost is not only in writing it, but also in people needing to read it, or people learning that they don't need to read it because a gadget is not useful to them.

One problem with programmers, in general, is that they seem to be rugged individualist, and yet there is little constraint on them. Ask an engineer to build an electronics module and he is constrained by choice of chassis and integrated circuit family. Constraining a programmer to use FORTRAN, on the other hand is almost no constraint at all. Try to constrain a programmer to write his source files to certain standards and you'll find that if he doesn't like it then he may either ignore it, or complain bitterly about it. And yet, code written by one person, needs to be integrated with code written by others, and may need to be maintained at a much later time by another.

Another area that sometimes leads to difficulty is how of the software development teams are organized. Ideally, one would like to see a clear chain of command from top level manager to individual software writers. Frequently, however, one finds a set of people from the various collaborating institutions, and at various stages in their professional career. The software manager in HEP thus doesn't necessarily have the same level of control, authority, or influence over the software team as say a manager would in industry. When a programmer doesn't agree on development directions with his manager, and the two are from different institutions there can be additional problems. He may try to circumvent his manager by going to his manager's manager, or by going to his real boss (i.e. the person who effectively signs his pay check) That person may not even be a software person. Such situations are probably not the cause of the software problems in HEP, but when other problems exist, these situations surely don't help.

The software team structure leads to another area: how are decisions made? One needs to understand the consequences of decisions made by committee, by recognized leaders, or by who ever is doing the work. Also an important question is how long decisions last. On the one hand one would like to have strong leadership in order that the software efforts lead to a coherent, well engineered whole. This works if the person in charge is a respected, competent leader. It doesn't work when the person in charge is a leader on an organization chart only and he isn't recognized as competent by the software team. On the other hand, a committee decision can function well by bringing out the best ideas from all the committee members. Or a committee can make decisions based on who talks the loudest or who is most persistent in pushing his ideas. The committee room could be the battle ground of religious wars where compromises are made in the heat of battle. Some of these may come back later to be the source of many problems. The person writing a software subsystem can be the best person to decide how it is going to look to the users, if he has a good feel for end-user needs, or he can be the worst person if he lacks that good feel.

Even with an attempt of strong leadership, one still needs to realize that these large software system are never built from scratch. This leads to code in different areas that could have quite different styles, internal rules, and methods. Temporary interfaces are made between these different areas which may never be eliminated. In practice, one may never be able to achieve a desired level of uniformity of the code.

Another question is what basic methodologies are being used. Some people argue for "old proven" methods. This is fine if they have indeed proven themselves to work well. But they made indeed be old and proven themselves to be a burden. Others argue for "modern engineering" methods. Some of these methods are indeed modern and effective, others may be a passing fad or a dream in computer scientist mind. One must keep in mind that some inventors of new methods are theorist who have lost contact with the real world, while others have gained insight from long experience on large software projects. Also is the question on whether new methods are tried out on a small scale to see how they work in practice, or whether decisions are made to use them first on a big scale. The word prototyping again comes to mind.

There is also question of overall project control. When spending large sums of money for a detector, it is not uncommon that the funding agency requires a rigorous form project control and hosts meetings to see that they are being met. What does software project control mean in HEP? There may be little control leading to a software free for all. Or there may be an overly rigid control creating a unnecessary burden on all the writers. One needs some control but only where it is cost-effective. One also needs to give programmers a lot of freedom because software is more of

an art form than a commodity that can be mass produced in a factory. And yet, all the code needs to fit into an integrated whole.

Project control is one item and quality control is another. There is some form of quality control being exercised by the software development team, whether the words are being used or not. It may be in the form of trusting the author of the code that it works, in one extreme, or code may go through regular technical reviews in another extreme. Some programmers can be trusted, while others need guidance. Technical peer reviews can be very effective in bringing out faults in a code or is basic design, or they can be a waste of time depending on how they are conducted.

In general, there is little professionalism in managing the software effort, compared to that found in the building of the hardware. All of the factors mentioned above, plus the independent mindedness of most HEP software writers, contribute to this lack of professionalism.

3.0 Software lifecycle

To properly manage any project, not only software, one needs to be able to monitor progress towards completion. To measure progress, one needs a specification or requirements document to know what completion means. Neither is easy to do with any software project and it seems especially hard for HEP software projects. It is much easier with hardware projects, where one can really see things being designed, built, tested, and installed.

The software lifecycle is much more difficult to visualize. Let us take a closer look at it. It must start with a planning stage. A software team is formed and responsibility for each sub-detector is dealt out. A sub-group may or may not be identified to deal with architecture, utilities and tools. The requirements document may or may not be written, but in many cases it might be just one sentence: write a Monte Carlo program, and a program to reconstruct the events. That might be the end of planning and the team then proceeds to the next step: writing code.

The code writing stages proceeds with apparent rapid progress. Thousands of lines of code are written by each sub-group each week. Measuring the number of lines of code written, however, is not a good measure of progress. The sub-systems must still be integrated into the whole, and code needs to be debugged. Here the 90-90 syndrome may happen. When asked for a progress report, the development team might say they are 90% done and should be completed in a few weeks. After that amount of time has elapsed the next progress report says that the code is 90% done and should be completed in a few weeks or more. The 90-90 syndrome is when code is 90% done about 90% of the elapsed time. It happens all too often with software projects and when it does it is a sure sign that one doesn't know how to measure progress.

Finally, at some point the whole code runs without bombing. Could one use it for production on real events to publish results? Probably not, it probably can't be trusted to do physics with reconstructed events.

The next stage is that of tuning the code to get it to work halfway decently. One needs to understand what's wrong with it. One probably needs to change the value of lots of cuts. Even some parts of the code will probably have to be rewritten. At some point one can ask again if it is ready to be used for production. The answer might be "not really", but if a conference is coming up in a few months, the group may have to use it for very preliminary results.

The next stage is fixing the major faults in the program. This may imply making some compromises. For example, track finding efficiency may be very high, but it consumes much to much CPU time, thus re-programming to reduce the CPU time at some cost to efficiency may be required. Other major faults may be things like too much memory is needed, or the interface between subsystems may be very wrong, or the wrong data may be on the Data Summary Tapes. When this stage is complete, is the program ready to use for production? The answer is of course that it will be used for production because one has done as best as one knows how. The step of tuning and fixing rarely ever stops until the detector is ready to be decommissioned.

The above description of the software lifecycle is admittedly oversimplified and somewhat cynical. For example, some large groups have gone much further at the planning stage by using formal methods to generate specifications. However, there is some truth in this description. In attempting to measure progress, are people fully counting the very long amount of time spent in tuning and fixing major bugs? Or do the time-line charts end when the code runs without bombing? Progress in software is hard to measure, but however we attempt to do it, we must taken into account

that code must not only be written but also meet some standards of quality and fit within some budget of CPU resources required.

4.0 Production Releases of Software

A major program is not written just once, but goes to releases with ever better versions. How are these releases managed? To answer this question, we need to look at one model of the stages the code goes through.

After code has been newly written or modified from previous version it is usually first tested in the private space of an individual. This must be so, because the code probably has numerous bugs at this stage and all other users need to be isolated from it.

Code than moves to a staging area I call development. This area is shared by all the members of the subgroup that need the latest version. If modularity of the program is good, this usually means only that subgroup uses the code.

The next stage doesn't generally exist in HEP, but I consider a very valuable stage. I call is pre-production. It should be an area where development code from the subgroups first come together before being released further. There could be a quality control team that tests the code in this area. We are all familiar with software companies that have beta release of new software. HEP generated code should also have this concept so a few can test code before it effects the whole group.

The final stage is production area. It is used not only for production jobs, but is the stable code being used by all the development teams. Many detector groups have a great deal of difficulty making a new production release. It is an area where the software problems in HEP are most visible. Frequently programs that used to work in development, now bomb when in production. They may be unexpected interactions between modules tested separately in their development areas.

An important question is how often to make production releases. Some groups release on a fixed interval between releases. One large group I've been told, has a new release monthly and with each release nobody's code runs without bombing for a few days to a week. Some groups find it so time consuming to do a new release that they do it rarely, thus, most of the time the production release is obsolete. Both of these extremes cases are symptoms of serious software problems. For example, there could be serious problems in the modularity of the code and/or with the quality control methods being used.

My own feeling on making releases is the following. It should not be so frequent that production code appears to be unstable and it should not be so infrequent that everybody is using all the development area. The time interval between releases will thus vary over the life of the detector. By observing what people are doing or feeling, one can judge if the cycle is too short or too long.

5.0 What Needs to be Done?

The question is, then, what do we need to do and what tools to we need in order to keep our software efforts from being such a mess? I don't pretend to know all the answers, but will mention three possibilities below.

First, a large software effort needs a good design. Good design comes with the proper modularity, which may not be as simple as division by detector type. Between the modules, there should be well designed interfaces, which usually come in the form of FORTRAN COMMON blocks and/or data banks. A good design of a large project can not be laid down correctly from the start; a certain amount of prototyping needs to be done. When a software team knows it is building a prototype, the whole attitude of approaching decisions changes to the better. That is, final decisions are not being made, only decisions tot test the prototype.

Second, the program has to have a good architecture and a good set of tools. These items are good if they are intuitive to use. The sign of bad architecture and tools is when users complain of poor documentation or too much documentation. The architecture and tools should not be overly constraining or overly protective. It is better to allow the user to make mistakes, provided he can find them easy, than to have a protective system which the user can't figure out how to use. There should be some uniformity of style across levels and not too many levels to learn. By a level I mean, for example, the batch control language level or the compiled source code level. I've heard of one group that requires the users to learn four levels, none of which are FORTRAN or the native batch language.

Third, as in any large project one needs to have progress and quality controls. Unlike hardware, it is much harder to quantify progress or quality with software. Although difficult, it is not impossible to invent some measurement tools, with which a software manager can judge the rate of progress. These tools may not be computer based. At the very least, peer review of software modules should be done systematically to judge progress and quality.

6.0 Conclusion

It is generally felt in our community, that with each generation large detector, software is becoming a bigger and bigger problem. If we extrapolate this trend to the SSC era, software would be a very big problem indeed. Many people are inventing or using new tools to attack the software problem, but some of these tools are like aspirins, they alleviate some pain but don't cure the disease. We need to understand the real causes of our current problems, before we can find the real solutions.

References

[1] Report of the Task Force on Detector R&D for the Superconducting Super Collider, SSC-SR-1021, June 1986

[2] J. Manzo, Computer Physics Comm., 45 (1987) 215.