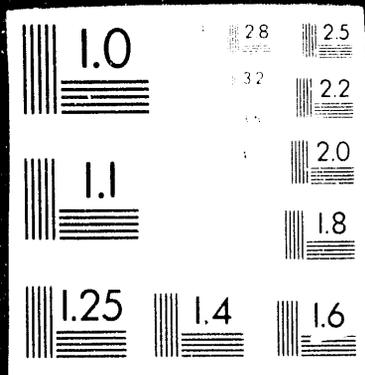


1 OF 1



LBL--32937

DE93 004767

**Software Engineering:
*What do experiments need?***

Stewart C. Loken

Information and Computing Sciences Division
Lawrence Berkeley Laboratory
Berkeley, CA 94720

This work was supported by the Director, Office of Energy Research, Office of the Scientific Computing Staff,
of the U. S. Department of Energy under Contract No. DE-AC03-76F00098.

MASTER

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

Software Engineering: What do experiments need?

Stewart C. Loken

Information and Computing Sciences Division
Lawrence Berkeley Laboratory, Berkeley, CA 94720

This paper will review the use of software engineering in High Energy Physics experiments and will suggest ways in which that use will change significantly in the future. In Principle, experiments do not need software engineering. We can verify this by looking at a long series of experiments that have succeeded without it. Some experiments that have tried to apply formal engineering methods have given them up before the software was complete. Future experiments, however, face many new challenges in the larger data samples, larger and more distributed collaborations and longer time scales. In addition, new tools are available to aid in the development of complex software systems. It is appropriate, therefore, to examine whether there is now a better match between software engineering and the needs of experiments.

Introduction

Computing and, in particular, software are now recognized as critical to the success of High Energy Physics experiments. Despite this, there is still considerable controversy on how software systems should be designed and implemented. This is perhaps surprising given that software engineering seems to be universally accepted in most parts of computer-based industry.

When we ask whether experiments need software engineering, the answer seems to be no. We can look at a long series of successful experiments that have succeeded without any formal engineering practices. No experiment has ever failed because of its lack of software engineering or, at least, none will admit it.

The experiments have succeeded, in part, because the groups were small, the data samples were small and the computing was usually carried out on a single computer system. In addition, there were few choices that had to be made in terms of languages or tools.

Requirements for New Experiments

The complexity of modern experiments has led many people to conclude that a new approach is needed. Based on the perceived success of software engineering in other areas, some of these people have turned to software engineering.

The problems of the new experiments are a result of the fact that there are now much larger collaborations and many more people are involved in developing software. The collaborations are much more distributed than in the past and the experiments last much longer. The new facilities generate much more data at higher rates than older detectors. This means that computing systems and storage systems must be larger and more complex. Finally, the modern trend toward workstations has increased the need to integrate heterogeneous computer systems and networks to permit access to the data and computing systems.

To match these needs experiments need tools for managing software development by a large, dispersed group. They need to produce reliable code that runs on heterogeneous systems. To support the experiment over its history they need to develop maintenance and upgrade plans. They need to plan systems for data management and storage.

Searching for the Silver Bullet

When these problems were recognized in the mid-1980's software developers began to search for solutions that would give programs that were high-quality, reliable and maintainable without a great investment in effort. The first such solution was an established formalism known as Structured Analysis / Structured Design or SD/SD. This formalism provides tools to describe requirements, Data-flow Diagrams, Entity Relationship Diagrams and State-transition Diagrams as well as tools to aid in design, Structure Charts. Two experiments, ALEPH and D0 adopted the techniques and trained a number of scientists and programmers in the methods.

Over the years, however, the methodology has been dropped and code is now developed in the more traditional ways.

There are a number of reasons that these efforts did not fully succeed. There were few computer based tools to support the formal methodology. As a result it was hard to maintain the results of the analysis and design. More important, however, was the fact that the methods were based on an understanding of the basic process. Just as learning to draw a circuit diagram does not make one a circuit designer, learning to draw Data-flow diagrams does not immediately generate quality software. Unfortunately, thinking is still required.

Modern software now holds out the promise of more quick fixes to complicated problems. Perhaps the most widely embraced is Object Oriented Programming which offers the prospect of modular, reusable systems. Other examples include Interface builders, Computer-Aided Software Engineering (CASE) Tools, Distributed Computing Environment (DCE) and the Application Visualization System (AVS). Each of these may have a role in the development of software for new experiments. But again, we should not confuse the tools with the approach.

What is Engineering?

In describing the software development process as *engineering*, we are making the assumption that we can apply some objective criteria to measure the success of our efforts. In the case of civil engineering, we can determine how to build a stronger bridge. In mechanical engineering, we can measure a part to show that it will fit into some larger system. There are, however, no measures of software strength or size that are meaningful to evaluate the quality of the development process.

In trying to understand the meaning of engineering, it is still useful to examine the analogy with hardware systems. Taking the case of a High Energy Physics detector, we can start with a very abstract view of the system. We can ask what particles we need to detect and what detector components we need to do that. We can then ask how they must be laid out and what the interfaces between them must be.

Having done this we can look at single detector elements. With a calorimeter, for example, we can ask what resolution is required, or what interaction rate it must sustain. Having defined these requirements, we can ask what technologies will meet them. We can evaluate what the risks of each choice might be and how to minimize those risks. We can build and test prototypes to better understand how to reduce risks.

The detector design can be further refined by looking at issues related to maintenance. How do we fix components? Can the detector run with some components not working? How will we know that things are working? As the experiment evolves, we will need to change the hardware. What if the data rate goes up? Can we replace specific components with ones that work at higher rates. What technologies are likely to improve over the course of the experiment? Can new technologies be integrated easily?

All of these issues must be addressed in an engineering design for the detector. In engineering software, there are analogous issues that must be studied.

Engineering Software

Our goal in the engineering approach is to go from a concept to working software that can be maintained over the course of the experiment. To do this, it is traditional to look at the life cycle model for software development. We decompose the process into phases: Analysis (of requirements), Design, Implementation, Testing, and Maintenance (including modification).

In the analysis phase, we determine what the system must do and what it must provide to the users. How much data must it handle? How fast must the data move? To be certain that the requirements are correct, they must be validated by the users. This means that there must be reviews at all phases of the development, just as for hardware systems.

In the design phase, the developers must look at possible implementations. What are the possible approaches to meeting the requirements? What are the risks with each approach? Can we minimize the risks by prototyping the critical pieces? Again, reviews must ensure that the design is sound.

In the implementation phase the developer must produce working modules. These should take advantage of existing components and commercial systems where appropriate. By using external and internal standards, we can improve the chances that the modules will be useful as the system evolves. In the testing phase, we must verify that each module works as expected and then ensure that the system of modules functions correctly. The tests must be saved so that we can verify after each modification the no functionality has been changed.

Maintenance and modification is a critical issue for the large collaborations that last over many years. The software will be used at many sites and on many different computer systems. It must be updated on a regular basis and versions at different sites must track so that modules can be exchanged easily. In addition, new versions of the code must be integrated into systems as modules evolve.

Tools of the Trade

There are software engineering tools available to assist at every phase in the life cycle. These include CASE Tools, Programming Environments and Interface Builders. There are also tools for code maintenance and for regression testing. Many of these are described in other presentations to this conference.

It should be recognized, however, that the tools are not the engineering. They are just a technology choice that the experiment must make. In the end, the tools are only as good as the people using them. For a quality result, thinking is still required.

Conclusions

Experiments do need a rational approach to the overall computing problem. The approach must ensure that the computing system is designed to do the right job with technologies that minimize the risk of failure.

The real need in software engineering is not for a set of tools or languages. It is rather for an approach to understanding the software problem and developing the optimum solution based on the best available technology.

Acknowledgment

This work was supported by the Director, Office of Energy Research, Office of Scientific Computing Staff, of the U. S. Department of Energy under Contract No. DE-AC03-76F00098.

END

**DATE
FILMED**

2/17/93

