WHY DEVELOP PROGRAMS DIFFERENTLY THAN CURRENT HEP PRACTICE?

James T. Linnemann

Michigan State University

East Lansing, Michigan

USA

Abstract

Current industrial program development practice is reviewed and its relevance to high energy physics is discussed. The experience of DØ with these techniques is commented on, and a selected bibliography provided.

The motivation for the development of the approaches I will be discussing came from industry. In an environment in which it is easier to measure the relative costs of development and of maintenance, by counting the number of programmers involved in these tasks, it was realized that maintenance (modifying existing programs to meet new requirements, to adapt to new hardware, and to correct old problems) was consuming a very large fraction of the programming effort. Inquiry into the causes of the situation revealed that programs were typically inflexible (difficult to transport or modify), consisted of highly coupled parts (a change in one section affected other, not-obviously-connected sections) with long-range interactions (bugs and effects of modifications were not localized). All these problems of maintenance were exacerbated by poor documentation.

How is this experience relevant to high energy physics? One obvious translation is programmer cost to time--time needed for other efforts, or time to complete a project. A clear difference from the industrial environment is that we are typically both the users and developers--which helps the communication problems, but tempts us to be too easily content with sloppy definitions of the requirements of the software we intend to write. Ill-defined requirements have been found in industry to encourage time/cost over-runs, since changes become progressively more time-consuming to implement, the later in the life of the project they are attempted. It is more expensive to alter an actual product than a preliminary design. In addition, the nature of research guarantees that programs must change, so we cannot escape maintenance and should plan to make it as easy as possible. Further, the common situation of large experiments, in which program development is carried out in a distributed fashion, puts a very high premium on clear communication and precise specification. This sort of development is feasible only if the sub-projects undertaken at various institutions are clearly separable, and the links among the sub-projects are understood and made explicit.

All these indications point to the need for a more systematic planning process than has been typical of our normal pattern of program development, with much more time spent before actual coding begins in order to save on total development time. While we have not normally thought in terms of actually designing software, we are comfortable with the idea of spending a considerable amount of time designing hardware. Perhaps this is because we have the impression that hardware is much less tractable than software--but we should rethink this notion when contemplating projects with 100,000 or more lines of code and 50 or more authors. Note that design implies a considerable level of detail--an electronics design must specify the full organization and interconnection of all the chips used, and depends, for even this relative degree of compactness, on the existence of books specifying the **behaviour** of the chips (low-level building blocks) used.

Designing Software

Current industrial practice recognizes that it is reasonable to spend up to half of the total time, from start to finish of a project, before starting coding. Coding itself is not the most time-intensive remaining step; rather it is debugging. The objective of the planning phase is to minimize the debugging phase. For this, one needs a good design, not just the first thing that comes to mind. This implies the need for a low-investment prototype or paper-design that one can discard without regret if need be.

In making a design, one attempts to specify the system, and verify that it does what's needed. By considering the system as a whole (not "we'll add that later"), one can understand the interrelations of the parts and document them. This communication requires common recognized conventions; it also assumes that after having designed the program, the implementors will actually follow the blueprint, not making random enhancements without bothering to tell anyone of the changes of function. Studies of human perception lead one to prefer a graphical representation of the interrelations, and to try to hold the level of detail presented at any one time to a tolerable level: 1 page of code per subroutine, 3 levels of indirect addressing or loop nesting, and 7 +/-3 objects (routines, arguments, logical alternatives) which must be considered at any one time.

A good design from the point of view of total life-cycle cost (not just running time, but also people time to develop, debug and later understand and modify) will strive to cut down links between components. Subroutines should perform single identifiable functions, not a group of "similar" functions. They preferably should have no internal memory (do the same thing each time), have no side effects, and in general behave as a simple black box, whose function is easily understood and easily modified if need be. The communication among modules should be as explicit as possible, with information made available to a module only when needed. In a group of subroutines, control should be as localized as possible--passing of control information should be minimized, flags should be yes/no whenever possible, use of special data values to carry control information should be shunned, and errors should be handled as locally as possible.

Software engineering is the name of the field devoted to techniques aimed at achieving these goals. The Aleph experiment, DØ, SLD, Delphi, and Opal are all using these techniques at varying levels of formality. One of the many variants goes by the name of Structured Analysis/Structured Design (SA/SD). I will be describing these techniques shortly.

Another of the ideas of software engineering is the walkthrough, a public display of work associated with a program. The idea behind the walkthrough is that it is fairly difficult to see your own mistakes, but that trying to explain things to others makes this much easier. Some of us don't care to show our programs or plans to others. In a collaboration, however, a program is no longer private, because other people depend on it, and someone else will have to read it anyway--why not before it's had the chance to make them mad!! The general guidelines for walkthroughs include a group of 3-7 people, not going much beyond 1 hour, attacking the product and not the producer, sticking to a checklist of features to be looked over, and only pointing out errors, not trying to design by committee.

Outline of SA/SD

The term "structured" in the name refers to the use of a particular set of tools and procedures, chosen to have sufficient expressive power, to address relevant issues, and to have produced good designs. The analogous tools in digital electronics design are to block diagrams, logic schematics, truth tables, state diagrams, timing diagrams, and signal lists. The term "analysis" is used as in "systems analyst"--it refers to the process of discerning what exactly it is that the system must do--(its essence). The term "design" refers to deriving a blueprint for implementation--how to do it.

In SA/SD, one strives to separate the essence from the **implementation** . The essence assumes perfect technology internal to the system while recognizing imperfect technology outside the system; it tries to understand the functions of the system in terms of concepts of the work outside the system. This separation of essence and implementation is feasible since the implementation corresponds to a lower level of detail, and can be mapped into a lower position in the program heirarchy. Further, such a separation is desirable, since from a point of view of maintenance, seldom do essence and interfaces change at the same time. If they are separated in the program, one need be concerned with less code to make a given change. These separations do carry a penalty in overheads, typically found to be of the order of 10% due to extra subroutine calls. It is assumed that most programs do not go from useful to useless on the basis of a 10% change in running time.

Structured Analysis

An outline of the procedures of structure analysis are as follows:

First, one defines the requirements and constraints to be met.

Next, one models the environment of the system. This defines the scope in terms of what is inside and what is outside the system. This is shown on a context diagram (Figure 1). Things passing through this boundary define the interactions of the system with the outside world. From these a list of stimuli (or "events") to which the system must respond is drawn up.

Finally, one models the behaviours of the system. One defines processes which represent the responses to the external stimuli. These processes will communicate with the outside world via data flows (information "in motion"), and with each other by data stores (information "at rest") -- the stimuli are not normally synchronized with each other. Typical tools used at this stage are called data-flow diagrams (showing processes, data flows, and the data stores); entity-relationship diagrams (showing the interrelation of various data); and process mini-specifications (showing what the processes must do).

The data flow diagram is probably the single most useful tool (see figure 2). It takes a data-view of the system, and shows logical interdependencies of various processes (not time ordering, as in a flow chart). A process transforms input data into output data. Data flows coming from the outside world are the stimuli; data flows from processes to stores represent writes, modifies, or deletes of information in the stores, while flows coming from stores to processes represent reading all or part of the information in the store. Essential processes are responses to the external stimuli. In practice, usually one will need to refer to the data dictionary to ascertain that enough information is in fact available to a process to produce the output data.

Structured Design

At the stage of structured design, one begins to make decisions concerning implementation. First, one assigns processes (or parts of processes) to processors (computers or people). The tool here is a dataflow diagram, on which one scribbles. At a more detailed level, one assigns processes to tasks. If the assignment is nontrivial, one may wish to draw another dataflow diagram with processors or tasks as bubbles to expose the interfaces. The aims are to minimize interface complexity and traffic, minimize data duplication, and satisfy any externally imposed constraints. Avoiding splitting of essential processes usually helps attain these goals.

Next, one must finish the specification of the processes, starting from the mini-specs. One may have to specify the processes in more detail, and perhaps begin to show control information on the dataflow diagram. In addition, processes necessary for implementation but not for essence should be added at this time, as a sort of shell around the essential processes.

Examples of implementation processes are formatting, checking, or classifying.

Now, one derives a **hierarchy** from the dataflow network representation. The tool here is called a structure chart; it shows subroutine call **hierarchy**, as well as information passage among modules. There are various strategies for picking a good **hierarchical** structure.

One next revises the provisional design, using various heuristics which normally typify good designs; chief among these are high cohesion (modules which do single, well-defined functions) and low coupling (minimal data transfer among modules, and even less control information passage).

Then comes actual coding. Here the tools are the more familiar ideas of structured programming.

Next, one tests the code to expose the errors in it. There are errors--the goal is to find them and get rid of them. The main tools are this attitude, and a plan, including data designed to flush problems and the expected responses to the test data.

Finally, if the program does not meet the requirements defined for necessary speed, one optimizes. The most important optimization should already have been done at the design stage, in choice of data structure and algorithm. Fine tuning should be done only as needed. Get a spy program to find where the problem really is--normally 10% of the code takes up to 50% of execution time, so don't waste time optimizing outside that 10%.

What has the experience of applying these techniques been?

Unfortunately, there is no completed project to point to. Aleph has been using these techniques intensively for at least a year, and has had commercial courses given in the methods given to a substantial fraction of the collaboration. My group, DØ, has been using them for a shorter time, and only I have been sent to a commercial course.

In our experience, drawing good dataflow diagrams is hard--part of the problem is inexperience, but a lot of it is just understanding the interconnections. The dataflow diagrams and data dictionaries are useful. They greatly enhance communication, lend themselves to precision and verification, and offer the hope of high quality documentation.

Maintaining this documentation requires software support for large projects. To date, the main **contenders** are SATOOLS from Tektronics on VAX, and Analyst Toolkit from Yourdon, Inc. and PCSA from Structsoft, among others, on IBM PC and clones.

The techniques seem to match well to specification of online computing, and of the upper levels of offline; the middle levels of offline, where many alternatives exist, are not as well matched, but the lower levels again seem to be well described.

To learn the techniques, various alternatives are available. Commercial courses are most useful if tailored to the HEP environment by hiring the instructor as a consultant for a time; books are available and reasonably useful for motivated individuals. For those not willing to take that much time, introduction to the techniques through participation in walkthroughs is useful.

Conclusion

Before ending with a bibliography on the subject, I would like to offer a few intentionally provocative remarks.

Experimental High Energy Physicists are working far from the forefront of computer science developments. We behave as if we don't care whether our programs are correct or not. Correctness is more important than efficiency--correctness is efficiency when measured globally (do it right or do it over). Fortran 77 resists most of the techniques for producing more reliable and maintainable programs. Learning a second language is a real eye-opener in this regard--you begin to see the limitations and biases of your first language. Look at Modula, Pascal, Ada, or the proposed Fortran 8x.

We have tended to ignore other areas of programming expertise which may well be quite relevant to us--NASA, national fusion facilities, oil refinery control, accelerator control and expert systems may have things to teach us about offline or online computing.

These techniques were discovered through the pressure of large projects, but are certainly applicable at small-scale projects as well; it would be very useful to see them applied to smaller projects with faster turnaround times.

Bibliography

In the area of project management, Brooks' The Mythical Man Month and Metzger's Managing a Programming Project are quite useful in terms of ideas as well as practical suggestions. A general survey of software engineering can be found in Fairley's Software Engineering Concepts. Myers offers a sobering view of the Art of Software Testing. Serious testing is shown to be at least as hard as desinging the algorithm in the first place.

Structured Analysis and Design is well covered in Ward and Mellor's Structured Development for Real-time Systems (3 volumes, 450 pages or so); a good book on mainly structured design is Page-Jones' Practical Guide to Structured Systems Design. These two books are distributed by the Yourdon Press.

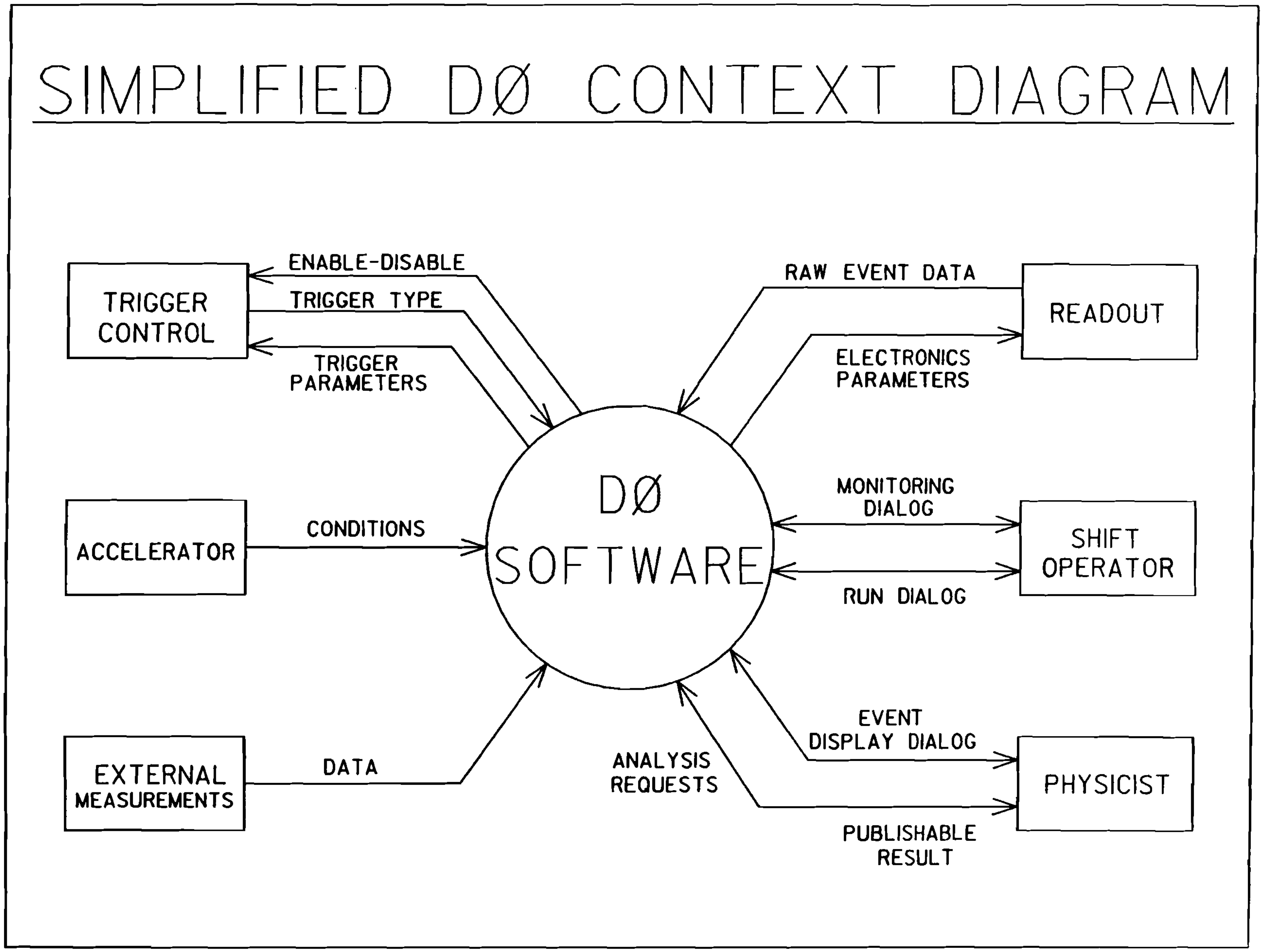
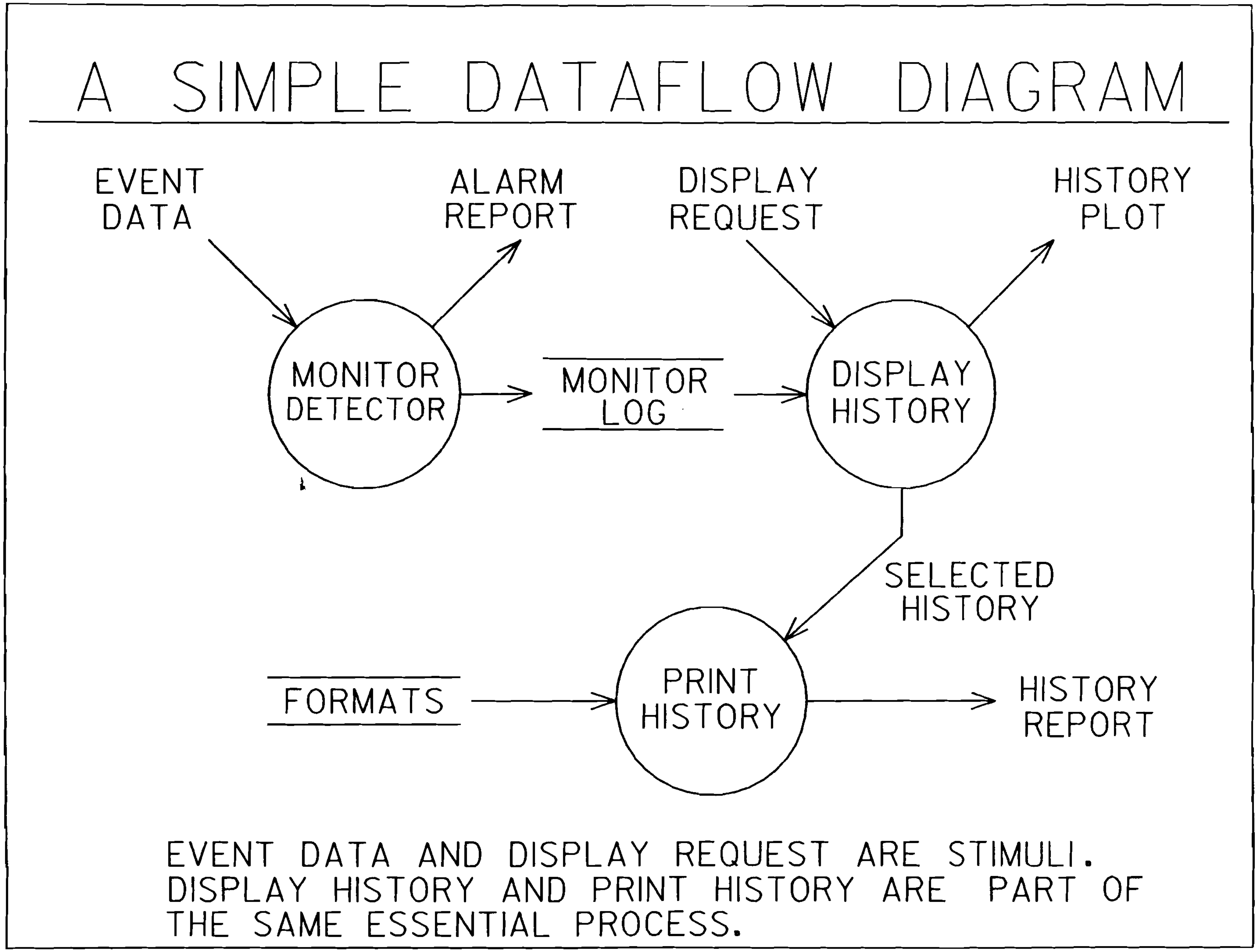

SIMPLIFIED DØ CONTEXT DIAGRAM
TRIGGER CONTROL
ENABLE-DISABLE
TRIGGER TYPE
TRIGGER PARAMETERS
RAW EVENT DATA
READOUT
ELECTRONICS PARAMETERS
DØ SOFTWARE
ACCELERATOR
CONDITIONS
MONITORING DIALOG
SHIFT OPERATOR
RUN DIALOG
EXTERNAL MEASUREMENTS
DATA
ANALYSIS REQUESTS
EVENT DISPLAY DIALOG
PHYSICIST
PUBLISHABLE RESULT

Figure 1



EVENT DATA AND DISPLAY REQUEST ARE STIMULI.
DISPLAY HISTORY AND PRINT HISTORY ARE PART OF
THE SAME ESSENTIAL PROCESS.

Figure 2