CCB
2-27-74
①

This article from Datamation is by someone from FDR — the name might be Moore. (It wasn't meant to be anonymous; that was accidental). A lot of people who knew me thought I wrote it. I wish I had!

I Particularly like his definition of a bad programmer. (My personal record is about 12 years.)

the forum

The Forum is offered for readers who want to express their opinion on any aspect of information processing. Your contributions are invited.

WHAT A PROGRAMMER DOES

It has been believed that a programmer occasionally writes code and gets it running on a computer, and that this is what he is paid for. In spite of his obvious inefficiency, no one else seems to do this work more effectively. However, his activity is still observed principally as loafing—a kind of ritual (like the British and teatime) which must be put up with.

Another view of what a programmer does addresses more constructively all that "wasted" time and ritual. To understand this view, we must see what it is the programmer is trying to build. We all have some idea of what a program is, but not very often do we see a program from its author's point of view. Whatever a program is, it exists as an organization of many pieces with a high degree of structural integrity, and it is stable in its environment. That environment includes the mechanisms of the program itself, as well as the input data. To establish this integrity is the usual debugging task.

However, there is more to the environment of our program. It must be stable with respect to the threat of operator misunderstandings, with respect to invalid data, and to changes in specifications, up to a point. Obviously, to develop a code structure that is stable in the face of all these unknowns is a very difficult task. As a matter of fact, we programmers all cheat. The structure we develop includes more than the running code, more than the symbolic code, or even the operator's guide, the maintenance guide, or the design guide. For in fact, in response to any serious breach of the program's integrity, a programmer will become involved, as part of the integral organization built by the original programmer. If one now looks closely, he can begin to recognize the intent of those steps in the ritual of programming.

program dybbuk

In this ritual, the programmer must clearly anticipate the sources of threats to his program. He must imagine techniques to deal with those threats, and anticipate the instabilities of those techniques. Of course, as the programmer has a mind trained in objectivity, he will realize that in fact he cannot scare off the dybbuk who already is making its home in his structure by the time debugging begins. It is at this point that the agony of programming begins. The structure is now nearly mortal, and its vitality may well be stolen by a form of the "enemy" which could and ought to have been anticipated by its creator. The enemy, the dybbuk, of course is entropy; it appears in many forms, and sometimes in many places simultaneously.

The program must not only have a static stability, in withstanding such things as invalid data, but must also further respond to dynamic changes in environment. Of these two types of stability the first can generally be provided by careful programming, provided the ultimate operating environment is reasonably organized. A program is rarely killed by a failure of static stability. The death usually occurs because of a failure in the program's mechanisms for maintaining stability, in its abilities to respond to environmental change.

One mechanism for maintaining stability is the maintenance programmer. The longevity of the program is therefore dependent on the capability, comprehension and intelligence of this person. But humans are not omniscient in comprehending programs. As a matter of fact one of the most difficult intellectual endeavors is the analysis and comprehension of an existing program structure. Thus the resistance of a program to unsettling forces is critically dependent on its structural clarity, as measured by the effort required to analyze and comprehend it.

We see, then, that the structure called a program consists of more than we first thought, and includes stabilizing mechanisms which are far broader than the code itself.

the traumatic periods

We should consider the traumatic periods of the life of a program. The terminal trauma of a program occurs when it is challenged by entropy beyond its capacity to adjust. Obviously every program will have, or has had, this trauma, after which it is of no use. Most programs go through at least two additional traumatic periods. The first is when the static stability is first being developed—that is, during debugging. The second is when the program is first placed into a real, instead of artificial, environment—during system testing.

In both of these periods a major problem is the exercise and further development of the stabilizing mechanisms which allow the program to run under extreme, unsettling entropic forces. In both of these periods the primary instrument of stability is the original programmer, who reacts toward an indication of disorganization (a "bug"). He may well have built into the program checks which aid him in identifying the disorder and its cause.

The ritual of programming is of great consequence because it deals with the communication between the

original program author and the programmer responsible for maintaining the structural integrity of the program. When these two are the same man, there is still a significant amount of communication between them; in a large program one cannot remember the whole program. The author must organize the program into concept units, and leave a clear trace of the organization, as well as the content, of each unit. Thus that part of the ritual which attempts to provide *the basis for comprehending* a program structure is, in fact, the part of the programming activity which *most directly supports* the vitality of the program, from debugging on.

In fact, experience generally shows that an intense concentration by the programmer in optimizing the comprehensibility of a program's structure pays off, not only in debugging, but also in ease of coding and program layout. If a program is structured so that it consists of a set of functions that can be understood separately, then the layout of the program should be obvious, or at least explainable. Furthermore, the coding effort should be uncluttered by considerations beyond the algorithms for the "current" function.

the primary role

Thus, a programmer develops his program so that a human can comprehend it. Initially this is merely self-protective, as he must understand the program enough to get into production. It would appear that all good programmers attempt to do this, whether they recognize it or not. Furthermore, no one has seen a program which the machine could not comprehend but which a human did. Not even bad programmers (those whose programs die young), write comprehensible code; if they did, their programs would survive and they would be better programmers. Both the value and quality of a programmer's work improve directly with the importance he places on communicating his program to a human, rather than merely to the machine.

A programmer does not primarily write code; rather, he primarily writes to another programmer about his problem solution. The understanding of this fact is a final step in his maturation as a technician.