CCB
2-27-74
C

This internal IBM memo is much later than the other material, but reflects the same theme. The specific occasion was the appearance of of a proposed set of standards for the Program Logic Manuals for IBM system software, which were to be the panacea for documentation. Alas, they were concerned almost wholly with form, not content, and the information content of manuals produced in accordance with these standards, while high, wouldn't be of much use to anyone. As I said in the last ¶, these were impressively well done — but in much the same sense that a detailed reproduction of "Washington Crossing the Delaware" executed in variously colored and sizes of buttons might be well done. Just irrelevant and only temporarily interesting.

I think the piece had some circulation within IBM; it's hard to tell. We both can guess how much effect it had on PLMs.

clb

October 20, 1967

Comments on "Proposed Standards for Program Logic Manuals"

It's hard to take exception to either the stated objectives of the proposed standard or the methods by which these are achieved. There are several points which deserve comment, however.

- The standards adequately describe how to document "conventional" programming techniques--flow charts, load modules, procedure calls, etc. However, programming techniques are getting more sophisticated all the time, and there may be difficulty adequately presenting these in the traditional format. An example which comes to mind is the use of re-entrant subroutines with multiple entry points, in which one or more subsequent entry points are determined by the routine itself as a result of processing performed at the current entry point. (This technique is often used to manage asynchronous i/o activities or to schedule in a multiprocessing environment.)

- The standards talk about "inputs," "processing," and "outputs" as if these determined exactly what a given piece of code did, and could be explicitly identified. Again, with advanced coding techniques, inputs and outputs may be determined by multi-level indirectness, or by temporal and spatial contextual information which requires extensive processing to make explicit. And a routine need do no processing at all, but merely invoke one or more of a great number of possible sub-processes. It is perfectly possible, and often profitable, to invoke a routine which "doesn't know what its inputs are, doesn't know what the processes it applies are, and doesn't know what its outputs are". This is particularly true as systems grow hierarchically, relying not on hardware for their execution, but on other system software. A low-level example that comes to mind is the internal use of interpretive techniques; there are many others.

• My concern is that there seems to be no real provision for documenting programs structured as I have outlined. This might not be so bad but for the feedback effect; a system designer, or development programmer, may tend to avoid such powerful techniques because of the difficulty in "PLMing" them, and we have the case of the documentation tail wagging the system design dog--clearly an undesirable effect, so far as I am concerned.

• On a different plane, however, I still have some serious reservations about the utility of the documentation as described. Consider, for a minute, when the PLM's are likely to be used; when the program doesn't work or when it is to be modified to make it do something else. If the program did exactly what the PLM said it did, there would be no need to consult the PLM at all. Although it sounds facetious, a more useful function of the PLM would be to describe not how the program works, but how the program fails. After all, this is what has probably happened, and is why the FE or SE is perusing the memory dump. To this end, the PLM needs to state, quite explicitly, under what conditions the program fails, and how this failure manifests itself. The sorts of things we need to know fall under the general headings of Constraints, Assumptions, Scope, Limitations, etc. Additionally, we need to know the response to unexpected inputs; the response if the routine is executed under the wrong circumstances; the symptoms expected as the result of various hardware failures, both hard and intermittent; and the responses which might result from failure of associated software.

• The PLM takes great pains to tell us how each routine or module integrates with all the other modules or routines. However, what we really need to know is how to isolate a given module from all the others, so that we can test it in an environment more tractable to analysis and tracking down trouble. This may necessitate the design and writing of a small auxiliary routine to create a data set of known format, for instance; or a routine to verify the outputs of the code in question, or even suggested additions or modifications to the routine itself. In a word, the PLM should include diagnostic techniques.

• It is well known that every code or program represents a compromise among many competing factors--storage and speed of execution are the canonical ones, but there are always many others involved. It's probably true that if a PLM isn't being consulted because the program has failed, it's being consulted as a guide for modification of the routine because the limitations and compromises originally adopted are no longer valid or optimum. The designer and development programmer have a clear obligation to make sure that these

constraints, and the means for changing them, are explicitly conveyed to the writer of the PLM. All of this can really be summed up under the heading "How to make the code do something else" and maybe even "Don't try to make it do thus-and-such; it looks as if it should be easy, but there are hidden pitfalls". Candor is rare, but it's vitally needed.

- Implicit in the above comments has been the assumption that a PLM serves three purposes: as a maintenance guide, as a modification guide, and as an archival, which is to say tutorial guide. Most attention has been given to the shortcomings of the proposed standards in the first two areas, as these are of immediate concern. However, the third function is of importance over the long-haul, as intimated in an early point, and, once again candor can be of great value. To serve this purpose, at least some attention should be given to presenting those things that were done wrong and those decisions that were in error, but which weren't discovered until the coding or checkout or integration was so far along that a change was impossible. This always occurs, to a degree, and results in patchwork features in the design or coding which work, but really aren't the right way to do things. These are often the vulnerable places in the program, and may even have been described as such. To avoid perpetuating the bad techniques, an exposition under the general heading "How it should be done if it were being done over again". Progress comes from recognizing failures as well as from recognizing successes.

- Let me emphasize in summary, that the document is an impressive piece of work and, in detail, extremely well done. PLM's will have to be useful, however, and I believe their utility could be improved markedly along the lines I have indicated. Probably what's really needed is an Alpha test and Beta test for PLM's to see if they really meet the needs of those who must rely on them.

C. L. Baker

CLB:kd

cc: R. W. Arnold
M. Dyer
F. A. Fay
R. D. Phillips
P. H. Trautwein