THE BELL SYSTEM TECHNICAL JOURNAL
SAFEGUARD SUPPLEMENT
Printed in U.S.A.

SAFEGUARD Data-Processing System:

An Experiment in Software Development

By R. D. FREEMAN

(Manuscript received January 3, 1975)

This paper describes a type of flowchart review used as a program-development technique in which each programmer is required to give a box-by-box explanation of a detailed flowchart of his program to a small group of critical colleagues. Such reviews appear to have caught all the major software design errors before the code was written. It also cut the software-development time by at least 25 percent, representing a return of at least 10:1 in terms of software-development time saved as a result of the week of the group's time spent in the flowchart review sessions.

I. INTRODUCTION

This paper describes a program-development technique used in the programming of the sensor (i.e., radar) control portion of the early-1973 release of the software used in the Meck test system. For this release, the sensor control was completely redesigned and reprogrammed. Reprogramming provided an opportunity to experiment with techniques in program development. Of the techniques that were tried, "flowchart reviews" had the largest effect on the development effort.

II. BACKGROUND

Sensor control serves as the software interface between the Central Logic and Control computer and the phased-array Missile Site Radar (MSR) at the Meck Island test site of Kwajalein Atoll in the central Pacific. The most complex job done by sensor control is to resolve conflicting requests for radar usage, e.g., target search and target track. This is accomplished by changing the time at which one of the requested MSR transmit/receive order pairs is executed by an amount small enough not to degrade the validity of the resulting data. Since it is naturally desirable to obtain the maximum amount of data from the radar, the rules for performing this radar-order-conflict resolution are inherently complex. The memorandum analyzing these rules is about

100 pages and demonstrates that the resulting radar-order-conflict-resolution algorithm meets all the system requirements.

As the test missions at Meck Island became more complex, they began to strain the original version of sensor control. There were problems in program execution time and also in the limitations of the radar-order-conflict-resolution algorithm designed into the original version. It was therefore decided that sensor control would be rewritten, essentially from scratch, using a new data structure and an improved conflict-resolution algorithm.

The development of the new sensor control required about six man-years of work, including algorithm design and analysis but excluding any detailed documentation that might be written in the future. For reasons that are mainly historical and beyond the control of the sensor control group, the programming was done in assembly language. Every few tenths of a millisecond of execution time was important. The new sensor control requires about 5000 lines of assembly language code (plus a somewhat larger number of comment lines) and executes in about half the processor time (about 1.5 to 2 ms) of the old sensor control.

III. THE PROCESS OF FLOWCHART REVIEW

During the reprogramming of sensor control, flowchart reviews were used to find software-design errors or possible improvements before the code was written. As a sensor control group policy, before coding was started, the programmer wrote very detailed flowcharts and data-set layouts.* The flowcharts were to be sufficiently detailed that, given the flowchart, coding the routines would be almost a mechanical process. In particular, every decision point and all possible branches of control were to be shown. On the average, there were fewer than a half-dozen lines of code per flowchart box. The data-set layouts were in complete detail, i.e., down to the level of the bit. Given these layouts, coding the data sets was strictly mechanical. There were no specific format requirements for the flowcharts and data-set layouts except that they be easy to read.

As soon as the flowchart and data-set layouts for an area were complete, a review meeting was held. These review meetings were always attended by the group supervisor and several group members. Those group members specially knowledgeable in the area covered in a particular flowchart review were specifically asked to attend. Other mem-

* A "data-set layout" is a pictorial representation of the structure of a data area. Fields within memory locations are shown left to right across the page and consecutive memory locations are shown top to bottom down the page.

bers of the group were encouraged to attend. Flowchart reviews were also open to anyone else who was interested but, in practice, no one outside the group chose to attend. Except for the supervisor, people attending had either given flowchart reviews themselves or were scheduled to give them. The programmer whose flowchart was being reviewed, therefore, had a technically critical, but sympathetic, audience. Although the discussion of technical alternatives sometimes grew quite spirited, criticism of a programmer's design was never sarcastic and there was no gloating when an error was discovered.

At the beginning of each review meeting, copies of the flowchart and data-set layouts were passed out to all participants. Copies were not passed out ahead of time, nor were they later given to anyone who missed the review, primarily because it was unlikely that they would be read.

Usually the programmer began the flowchart review by giving a brief overview of how his code was structured. No high-level flowcharts were used. However, it proved quite easy for a programmer to point out what sections of his detailed flowchart represented what major functions and, in effect, to create a high-level flowchart in the course of the discussion. If the data-set structure used by his program was at all complex, the programmer usually gave a summary of the data structure at this point, leaving the definition of the specific fields for later. Occasionally, there was some discussion of alternative data structures at this point. Usually, however, any alternatives to the data structure designed by the programmer were suggested during the detailed discussion of the flowchart. This was probably because the functional structure of the data base had been one of the earliest decisions made and was a basis for an improved radar-order-conflict-resolution algorithm.

After this overview had been completed, the programmer explained his flowchart in detail. This explanation consisted simply of starting at the beginning and going through it box by box in the same order as the code they represented would be executed. If the descriptive phrase enclosed by a box was not self-explanatory, the programmer gave a brief explanation of what the code would do. For a few more complex algorithms, the programmer set up an example on the blackboard and carried it through during the discussion of the flowchart. The flowcharts were sufficiently detailed so that it was not necessary to describe how the code represented by a box in the flowchart would do the specified function; this was self-evident. However, it was often necessary to stop after reviewing all the individual boxes associated with a particular major function and to discuss whether the design represented by the flowchart would in fact carry out the desired function for any valid input and retain sanity for all possible inputs passed to sensor control. Also, the participants in the flowchart review interrupted the programmer with a question or comment on the average of once for every two to three boxes in the flowchart.

The participants in the flowchart review, although sympathetic, were expected to take an aggressive "I'm from Missouri and you have to prove it to me" attitude toward every assertion that the programmer made. If the programmer said that a data field began at a particular bit in a particular word, more than half the participants would turn to their data-set layouts to verify that. If the programmer said that the various inputs to a given internal subroutine could be divided into three classes, the other participants would try to think of a fourth. If the programmer said that the inputs from another module were in a particular format, the person responsible for that module would be asked to verify this. If it could not be verified on the spot, e.g., because the module owner was not present, it would be checked later. This aggressive questioning of the programmer's every assumption by his colleagues was undoubtedly the key to the success of these flowchart reviews. The programmer would sometimes catch a minor error, e.g., branch conditions reversed for a decision point, as he explained his flowchart to the group. However, the more significant problems were almost invariably found by the other participants.

Discovery of many more significant problems found during these flowchart reviews often resembled the way a lawyer sometimes (at least, on television) finds a major flaw in a witness's story during cross-examination. Instead of anyone at first noticing the basic problem with the design, someone would notice a minor problem. Two or three people, including the programmer responsible for the code, would then propose obvious patches to the design to handle this special case. The discussion of this minor problem would, however, have focused the group's attention on that particular area of the design. During the discussion of the best way to patch the design to handle this minor problem, someone would notice a second problem. Now that the group had seen two problems related to the same aspect of the design, comments would come thick and fast, with interruptions every few sentences. In a few minutes, this whole area of the design would be thoroughly explored and any problems would be obvious. Often, the person who noticed the second minor problem, and hence triggered the discussion leading to the discovery of the basic problem, was neither the person who noticed the first problem nor the programmer responsible for the code.

Another interesting feature of these flowchart reviews is the way two or three people, usually including the programmer responsible for the code, would occasionally seize the conversational initiative and draw the group down a side path. These side paths would often explore an alternative design in a manner not unlike a chess player exploring the consequences of a particular move. One person would suggest a modification to the original design; a second person might suggest that if you were going to make the first change, the design could then be improved by changing another feature. Another person might then suggest a third change, or might suggest that if you were already going far enough to make the first two changes, you could go all the way, make a certain change in one of the basic design assumptions and redo a portion of the design. These side paths were particularly useful in finding simplifications to the original design. In at least some cases, a few minutes of discussion saved a few weeks of programming and unit testing. In one area, the code used to recover from machine interrupts, the side path led to a spirited technical argument extending through several flowchart reviews and ultimately resulting in a design with more capabilities than any initial proposal.

As a conclusion to this description of the process of flowchart review, it is worth reemphasizing the importance of maintaining a matter-of-fact and unemotional atmosphere. This is essential so that the programmer can accept his colleagues' aggressive questioning as just the rules of the game. Viewed in that light, a flowchart review is just a form of professional review that is part of the programmer's job as a technical professional. A group of programmers meeting for a flowchart review is then not unlike M.D.s holding a seminar to discuss a particular patient's history and the treatment that is or was being given to him. However, if a matter-of-fact atmosphere were not maintained, the aggressive questioning in a flowchart review would be an intolerable insult to the programmer's pride as a technical professional.

IV. RESULTS OF UTILIZING FLOWCHART REVIEW

About two dozen flowchart reviews, including repeats, were required for all sensor control. Although the length of the flowchart reviews varied considerably, they averaged about two hours. Since the entire group often did not attend a flowchart review, two dozen two-hour reviews amounted to slightly less than a week of the group's time. As one would expect, the number of problems uncovered at the flowchart reviews varied considerably. However, the average two-hour flowchart review led to the discovery of about a dozen problems, varying in importance from trivial to major. As a result of the reviews, several areas of the new sensor control were redesigned essentially from scratch, several areas were changed significantly, and no area was left unchanged. Perhaps the best indication of the number of changes that resulted is the number of times that it was worthwhile to repeat the review. In roughly half the cases, the first flowchart review led to sufficiently extensive changes that a second review was held after the design had been modified. Had all the errors uncovered in the flowchart reviews been found a few at a time as the code was written and tested, it would easily have required at least several more months of the entire group's time (equal to roughly one-third of the time actually required) to complete the development of the new sensor control. Thus, assuming that the programmers would write detailed flowcharts or do some other form of detailed design for their own use, there was a return of over 10:1 on the week's worth of the group's time spent in the flowchart reviews. These calculations exclude the time saved in system testing by delivering higher quality software, which probably exceeds that saved during program development. The group responsible for programming the target search and target track algorithms used in the Meck test system has also used flowchart reviews like those used by the sensor control group, with similar results.

Perhaps the most striking result of using flowchart reviews was that all the major software design errors appear to have been caught during the reviews, before the code was written. Excluding a few cases where changes in the system requirements or the discovery of errors in engineering assumptions used by the sensor control group forced some redesign, the design was very stable after the completion of the flowchart review. This illustrates both one of the successful results of flowchart reviews and one of the chief limitations found. If the functional requirements and engineering algorithms remained stable, then the software design remained stable after the flowchart review. However, the flowchart reviews were not very useful in protecting against unexpected changes in system requirements or errors in clearly articulated—but wrong—engineering assumptions made by the entire group. Fortunately, there was only one case where this problem caused a large amount of redesign, and in that case the redesign occurred before any code had been written.

The use of flowchart reviews led to the discovery of two disadvantages that seem inherent in any such highly detailed review procedure. The first is that it involves too much detail to be useful during the preliminary design stage. Sensor control was a modestly sized set of programs designed by a small group whose desks were only a few steps from each other. Thus, the lack of a formal review process during the early stages of the design was not a real problem. However, looking back, it seems that some effort might have been saved if a more formal top-down design approach, with design reviews at intermediate points, had been adopted after the basics of the data structure and radar-order-conflict-resolution algorithm had been determined.

The second disadvantage is the level of boredom that must be tolerated. Interest drops off rapidly if no serious questions have been raised for 15 to 20 minutes, and the discussion becomes very boring. During the sensor control flowchart reviews, periods of intense boredom sometimes lasted over half an hour. Also, the policy of aggressively questioning every assertion sometimes leads to three- to five-minute discussions to resolve trivial points. Despite the boredom involved in this nit-picking, such discussions should not be dropped. Discovery of many major problems resulted from unsuccessful attempts to satisfactorily resolve what seemed at first to be trivial questions.

Concerning the amount of boredom that has to be tolerated during a flowchart review, experience throughout the flowchart review has been that if the leader does not care enough to personally take part in the flowchart reviews, they will not be held. If the leader of a group lets boredom take the edge off his personal aggressiveness, then the whole group loses its aggressiveness. Although it is hoped that the leader would be a key technical contributor to the review process, his chief responsibility is to maintain the group's aggressiveness despite the inevitable boredom—and the leader's personal example is critical in carrying out this responsibility.

To be sure, it is difficult for a supervisor to allocate the several hours required to take part in a flowchart review. However, if a fair-sized piece of software is being built, then the quality of the software design is an important factor in determining the quality of the supervisory group's output. Thus, ensuring the quality of the software design—by one method or another—is an important part of the supervisor's job.

Besides the group leader's personal example, motivating the group members to participate actively requires convincing them that the reviews are productive. Because of the number of problems found during the sensor control flowchart reviews, their usefulness was obvious to the participants, although no one—especially not the group supervisor—pretended that they were fun. As mentioned above, only those group members who had the knowledge to make a meaningful contribution to a particular flowchart review or who could learn from it were specifically asked to attend the review. No one was ever asked to participate in a flowchart review just because of arbitrary group rules. Those group members who were asked to attend, especially the lead programmers who were asked to participate in most of the reviews, were told frankly that the supervisor realized that this was not one of the more enjoyable parts of their job, but that they were being invited because their participation was important. In practice, there was no problem motivating the lead programmers to participate in so many flowchart reviews. The same personality traits that made a person into a lead programmer in the first place also made that person willing to put up with some boredom to obtain the satisfaction of having had a strong personal impact on the quality of the group's work.

V. COMPARISON WITH OTHER FORMS OF PROFESSIONAL REVIEW

It is worthwhile to compare the formal group-meeting style of flowchart review used in the development of the new sensor control with other forms of professional review that have been discussed in the literature. Flowchart reviews are very similar in spirit to Weinberg's concept of "egoless programming,"[1] in which programmers are trained to encourage other members of their programming team to contribute to their work; e.g., by reading their programs. The intent of egoless programming is for each program to be—as much as is practical—the product of the collective efforts of a programming team rather than the product of an individual programmer working in isolation (hence the term "egoless"). The group members are encouraged to be technically aggressive in reviewing each other's work. Also, as with flowchart reviews, group members are encouraged to be as matter-of-fact and unemotional as possible in pointing out errors or making suggestions. As Weinberg has reported, egoless programming has worked extremely well in some programming groups. One advantage of flowchart reviews compared with egoless programming is that flowchart reviews are a formal group meeting in which the supervisor takes part. Thus, their success is less dependent upon personalities and it is considerably easier for the supervisor to ensure that the reviews maintain a uniform standard of thoroughness.

Mills[2–4] has made several very innovative proposals [e.g., chief programmer teams, programmer librarians, top-down design/structured programming, PIDGIN (roughly similar to outlines)] as alternatives to flowcharts for organizing software development. See also the papers by Donaldson,[5] Miller,[6] Baker,[7,8] and Nichols.[9] Chief programmer teams, especially when combined with top-down design, provide an opportunity for a great deal of professional review.

Another technique similar to flowchart review is a "walk-through";[10] Mauceri[11] has used group meetings to walk through the actual code in a manner similar to the way that detailed flowcharts were reviewed in the development of the new sensor control. One difference between the work reported by Mauceri and the flowchart reviews used in the development of sensor control is the handling of problems discovered during the course of a review session. Mauceri reported that the procedure his groups used was to resolve questions and problems "off-line"; i.e., they were put on a list to be settled later. During the sensor control flowchart reviews, the questions and problems could be said to have been resolved "on-line"; that is, resolved immediately as they came up during the review if this were at all possible. As mentioned above, many major problems found during the flowchart reviews were discovered as a result of repeatedly unsuccessful attempts to resolve what first seemed to be trivial problems. Another difference between the sensor control flowchart reviews and the reviews reported by Mauceri is the inclusion of unit and module test cases in the reviews reported by Mauceri. This was not done in the development of the new sensor control. Instead, professional review of unit test cases was obtained by a technique suggested by the various experiments on code reading. Based on the detailed flowcharts used in the flowchart reviews, one senior person in the group did the functional design of the unit test cases for all of sensor control. The individual programmers were still responsible for unit testing of their own code. Thus, they had to review the proposed unit test cases for completeness and possible redundancy. In this way, unit test cases were examined in detail by two people.

It is interesting to compare the group-meeting-style flowchart reviews with the widely practiced technique of "code reading," in which a programmer's code is read line by line by either his manager or a senior programmer. In code reading, ideally the reviewer and the programmer read through the code together, although sometimes the programmer merely gives the reviewer a copy of his program listing. For code written in assembly language, the flowchart review has the advantage that it can be done earlier in the development cycle. However, if the code is to be written in one of the better high-level languages, it is not obvious that the professional review procedure should be based on flowcharts. Even if one were to use a formal group-meeting style of review, it might be better to skip writing highly detailed flowcharts and to base the review on the actual code as Mauceri and his colleagues did. One disadvantage of flowchart review compared with code reading is that a flowchart review will not detect minor coding errors; e.g., misnamed variables.

The fact that a flowchart review involves a much larger number of people than a typically two-person code-reading session is both an advantage and a disadvantage. As a disadvantage, the more people involved in a given review session, the more of the group's time is consumed. As an advantage, a group review appears to be able to detect many more errors, especially errors of omission (e.g., simply forgetting a given situation or a given class of inputs) than would be found if the design were reviewed by any single person. One of the more interesting features of the flowchart reviews was the fact that no one participant noticed half the errors that were found. This illustrates the advantage of flowchart reviews by a group of a programmer's colleagues, as compared with the more traditional managerial practice in which a programmer reviews his design, probably briefly, only with his manager. The traditional managerial review procedure is probably inferior to almost any reasonable procedure that involves the detailed review of a programmer's work by a group of his colleagues.

This is not to say that a formal group-meeting-style flowchart review is always to be preferred to code reading. Flowchart reviews are not very useful for small changes to existing code corresponding to less than several dozen lines of assembly language code and to flowcharts with fewer than a half-dozen boxes. Unless it is possible to review several such changes in one session, the flowchart review will probably be finished—accompanied by much grumbling by the participants whose work was interrupted—in about as much time as the people could be brought together. In fact, some months after the original version of sensor control was delivered to the system-integration team, code reading was introduced into the sensor control group to help tighten coding and testing of the minor changes being added to the original design. How this came to pass is a story with a useful moral.

Some months after the new sensor control was delivered to the system-integration team, minor additions had to be made to the code to provide for some new capabilities. These additions went beyond the software design that had been covered in the flowchart reviews. In the time since the new sensor control had been turned over to the system-integration team, the group, or at least the supervisor (the author), had grown too cocky. The code had run well during the several months of system-integration testing, and a series of minor changes had already been introduced with few problems. Probably significantly, the design for this first series of minor changes had been included in the original flowchart reviews; the implementation of these changes had been delayed. The new changes that went beyond the original design seemed at the time to be just more minor changes; no special review seemed needed. The programmers individually tested their code and released it after they felt that the changes had been thoroughly tested. Suddenly, during one week, the system-integration team found bugs in minor changes submitted by more than half the group. As a result of this, the supervisor got a useful lesson in humility and a certain amount of cheerful harassment from the system-integration team. To deal with this minor fiasco, a referee system was set up for minor changes. Each programmer submitting a minor change was required to select a referee from among the senior members of the group. The programmer would discuss both his proposed change and the procedure to be used in testing the change with the referee. The change could not be released until the referee was satisfied with the testing as well as with the code itself. After the referee system was introduced, the problem of bugs in minor changes came to a very satisfying end.

VI. LESSONS LEARNED

One lesson that was learned from the experiments described above is the extent of the increase in quality and productivity that can be obtained from the disciplined use of professional review. The use of flowchart reviews in the development of the new sensor control:

(*i*) Improved and simplified the software design.
(*ii*) Appears to have caught all the major software design errors before code was written.
(*iii*) Reduced the software development time by at least 25 percent.
(*iv*) Improved the quality of the software delivered.

The use of a referee procedure brought an end to the errors in minor changes turned over to the system-integration team. Other forms of professional review have led to similar results.

A second significant lesson can be learned by comparing professional review with some other techniques that have also led to improvements in program quality and programmer productivity; e.g., programming teams, modular and top-down design, and structured programming. A common denominator to these techniques is the increased structure and discipline placed on the process of writing software. Although what we now know about writing software is undoubtedly much less than what remains to be learned, it is already clear that designing and writing software needs to be a much more structured process than it is today.

REFERENCES

1. G. M. Weinberg, *The Psychology of Computer Programming*, New York: Van Nostrand Reinhold, 1971, pp. 56–64.
2. H. D. Mills, "Chief Programmer Teams: Principles and Procedures," Report N. FSC 71-5108, IBM Federal Systems Division, Gaithersburg, Maryland, 1971.
3. H. D. Mills, "Top-Down Programming in Large Systems," *Debugging Techniques in Large Systems*, R. Rustin, ed., Englewood Cliffs, New Jersey: Prentice-Hall, 1971, pp. 41–55.
4. F. T. Baker and H. D. Mills, "Chief Programmer Teams," Datamation, *19*, No. 12 (December 1973), pp. 58–61.
5. J. R. Donaldson, "Structured Programming," Datamation, *19*, No. 12 (December 1973), pp. 52–54.
6. E. F. Miller, Jr. and G. E. Lindamood, "Structured Programming Top-down Approach," Datamation, *19*, No. 12 (December 1973), pp. 55–57.
7. F. T. Baker, "Chief Programmer Team Management of Production Programming," IBM Systems Journal, *11*, No. 1 (January 1972), pp. 56–73.

8. F. T. Baker, "System Quality Through Structured Programming," Proc. AFIPS FJCC, *41*, Part I (1972), pp. 339–343.
9. B. C. Nichols, "SAFEGUARD Data-Processing System: Structured Programming and Programming Production Librarians," B.S.T.J., this issue, pp. S211–S219.
10. A. L. Scherr, "Developing and Testing a Large Programming System, OS/360 Time-Sharing Option," *Program Test Methods*, Englewood Cliffs: Prentice-Hall, 1973, pp. 165–180.
11. R. Mauceri, "Increasing Quality and Productivity Through the Development Process," speech given at Bell Laboratories, Madison, N.J., July 14, 1973.