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

SAFEGUARD Data-Processing System:

Structured Programming and Program Production Librarians

By B. C. NICHOLS

(Manuscript received January 3, 1975)

This paper discusses the phased implementation of structured programming techniques over a period of two years. It was observed that, by standardizing programming techniques, the resulting program becomes more maintainable and programmer productivity increases. By confining the clerical work of programming to the program librarian, productivity again increases.

I. INTRODUCTION

Structured programming techniques have been widely publicized throughout the data-processing industry. In March 1970, one programming department was chosen as a pilot group to test the validity of these techniques in the SAFEGUARD environment. This paper summarizes the experience gained in the ensuing two years, as increasingly advanced structuring techniques were used by the pilot group. Phased introduction of each technique is discussed to indicate that the transition from a conventional to a structured environment can be accomplished smoothly. Effects of the phased transition on personnel are discussed, and quantitative productivity data are provided for each phase. Although the statistical validity of these data must be qualified, a definite trend toward increased productivity is indicated.

II. DEFINITION OF TERMS

Within the pilot group, the term "structured programming" was used to identify five distinct techniques. They are structured code, top-down programming, code reading, PIDGIN, and the Program Production Library (PPL).

Structured code is based on a mathematical theorem that shows that any program can be developed by the appropriate nesting of three

basic logic patterns: sequence of operations, conditional branch to one of two operations, and repetition of an operation while a condition is true.[1] Elaboration of these patterns leads to the five basic logic structures used by the group to implement structured code: sequence, IFTHENELSE, DOWHILE, DOUNTIL, and CASE. Since only this statement grouping was permitted, standardization of code resulted. Also, adherence to these logic patterns results in complete control of all branching logic and therefore programs are easily readable from top to bottom.

Top-down programming requires that both design and code be developed from the control logic level down to the detail logic level. Program design has traditionally followed this approach, proceeding from system specifications to design instructions. Top-down design adds to this requirement that the control levels be coded prior to completing the detail design of lower-level paths.

Conventional program code, however, frequently does not follow the top-down approach. Detail level logic is often coded concurrently or before high-level control logic. Top-down code dictates that the next level of program code cannot be developed until all paths upon which this code depends have been coded and (preferably) tested.

Another structured technique, *code reading*, was made possible by the use of top-down programming and structured code. This is the practice of having all programmers exchange listings to ensure that each program is read by someone other than the author. Desk debugging is significantly increased and fewer, if any, preliminary clean-up computer runs become necessary.

Large programs, however, still present a problem since the ability to read them top to bottom is jeopardized by their total length. To resolve this problem, the pilot group used a segmenting technique that breaks down the program structure into functional segments. Each segment is then constructed so that ideally it occupies no more than one page of a program listing.

PIDGIN is a program design language that combines English, a programming language, and the structured conventions. This language was used to describe each functional segment. Through this design medium, system functions are visually broken into dependent segments showing the relation of each segment to the overall purpose of the program.

The last technique used in this study was the *Program Production Library* (PPL). The PPL concept stems from the observation that much of the task of computer programming is clerical. The PPL provided a standardized means for recording, cataloging, and filing all code generated, and it ensured a coherent library control system during program development and maintenance. It also provided a means for

standardizing the JCL-type interface to the computer whereby processing options (Compile, Linkedit, Modify, etc.) were invoked through key words chosen by the programmers. A more detailed discussion of the PPL appears in Section VII.

III. PROGRAMMING ENVIRONMENT

Structured programming techniques, PPL, and program librarians were introduced into a programming project over a one-year period and observed for an additional year. The project comprised the development of independent functional tests for the CLC operating system.[2] The complete set of tests was developed incrementally over several years, and the end product was a set of test specifications and the programs implementing them. Data for this study were gathered from the development of the test monitor facility and the first 11 test sets. The test monitor facility provided standard result recording for all tests, such that each test set contained no reused design or code. The SAFEGUARD assembler level language was used for all coding.

The nature of the development environment is also important for interpreting the results of this study. Test sets were being developed in parallel with the operating system. The CLC was the target computer,[3] but all software development occurred on the IBM 360, testing being accomplished on the CLC or by simulation on the 360.

The activities of the pilot project group were confined solely to test design, coding, and documentation. Testing and debugging were accomplished by a separate test team. However, correction of implementation and coding errors in response to error reports made by the test group was a continuing background activity to all development efforts. This maintenance activity reached a peak during the first two months following delivery of each test set.

A programming team consisting of three to four people, each having an average of two years of programming experience, was assigned to each test set. The schedule time allowed for the development of a test set was four to five months, or an average of 16 man-months. Each development cycle had three stages: test specification (2 man-months), test design (3 man-months), coding and documentation (11 man-months).

Another equally important aspect of the development environment was the personnel skill mix. The SAFEGUARD software proved to be a great equalizer in that personnel new to the project had to learn not only the complex application area, but also a new spectrum of support software. The result was that experience with SAFEGUARD software was frequently equal in value to overall programming experience. The rotation of SAFEGUARD-experienced personnel to related critical project

areas was common. In the pilot group, personnel assignments were rotated frequently throughout the two-year period studied, thus keeping the average programmer experience constant through each development cycle. Over a three-year period, a total of 21 programmers were assigned to the pilot group. Its total size ranged from 7 to 10.

The difficulty of the programming job is another important consideration. In retrospect, the tests performed in earlier sets are less complex but, at the time of their development, user documentation for the operating system was incomplete. The complexity of the later test sets was significantly higher; however, by this time documentation had improved, familiarization with the general modus operandi of the operating system had occurred, and personnel were accustomed to the test monitor interface. Hence, the relative difficulty of the programming job remained constant.

IV. IMPLEMENTATION PHASES

The test monitor and the first test set were developed using conventional programming methods. Improved programming techniques were then introduced in two distinct phases. The next three test sets were developed using structured programming and represent phase I. The next six were developed using structured programming, the PPL, and program librarians, representing phase II.

V. QUANTITATIVE RESULTS

Table I quantifies the effect of each phase on programmer productivity over the two-year period. For this study, productivity is defined as the number of delivered lines of code produced per day during the coding phase. Activities during coding include design, documentation, coding of the unit programs, and maintenance of previous test sets. Debugging was not a part of this activity, as has been previously discussed. Source statement counts include all lines coded, including comments and other descriptive lines required to meet documentation standards. The object size includes both instruction and data areas and measures the delivered product, as do the source lines. The ratio of source to object is provided to give the reader a rough indication of the number of executable instructions per line coded. Programmer-days reflects cumulative elapsed days for each programmer, and it does not account for overtime, vacations of less then one week, or illness. Also, it reflects only days spent by programmers, i.e., it does not include management or program librarians. It is the intent of this study to indicate the effect of new technologies on programmer productivity as defined above, rather than on overall product cost.

Table I — Comparison of productivity

Delivered Item	Source Lines	Object Size (32-bit words)	Ratio of Source to Object Size	Programmer-Days (Coding Phase)	Source Lines Per Programmer-Day
		Conventional			
Test Monitor	4056	1914	2.1	301	13
Set 1	6072	6540	0.9	381	16
		Phase I			
Set 2	9654	7300	1.3	240	40
Set 3	4271	2150	2.0	150	28
Set 4	6601	3500	1.9	130	51
		Phase II			
Set 5	9968	3700	2.7	165	60
Set 6	14689	7000	2.1	225	65
Set 7	16773	6500	2.6	150	111
Set 8	5588	3900	1.4	136	41
Set 9	11666	5830	2.0	160	73
Set 10	11596	6230	1.9	158	74

A comparison of raw productivity rates was made over the two-year period reported. No difference between the three- or four-person team was observed, and thus no distinction is made in Table I. The data in Table I should not be used out of the context of the background already provided in previous sections, since this can lead to rather startling conclusions. Table II summarizes the data for each phase, but must only be considered as indicating a trend rather than actual percentage gains. The productivity figures reported are dependent on many factors unique to the specific development environment of this study.

VI. PHASE I

Phase I introduced structured programming, code reading, and unit-level top-down approach into the development cycle. These techniques can probably be introduced into any existing programming project if the following prerequisites are satisfied. The programming language in

Table II — Summary of results

Implementation Phase	Total Source Lines	Total Programmer-Days	Average Lines per Day
Conventional	10128	682	14.7
Phase I	20526	520	39.8
Phase II	70280	994	70.8

use must include instructions that implement the structured programming logic patterns. This may require the development of a special set of macros to support the branching logic. In the case of the project being studied, two man-months were required to develop a macro package to provide structured statements in the language used. At the outset of phased introduction, a programmer experienced in structured programming must be available for consultation. This person need not be a member of the group itself, but should conduct an orientation seminar for those programmers asked to use the new techniques. The program areas selected for structured programming must be functionally separate from other areas. It is difficult to introduce these techniques into an existing program unless the new code represents a distinct functional unit that can be restricted to having only one entry and one exit.

The effects of phase I implementation were significant. Resistance from programmers occurred at the orientation seminar and during the early stages of implementation. However, once they began to use structuring techniques for program control, acceptance was quick. Resistance to the new techniques seemed to be directly proportional to programming experience. That is, firmly established coding habits were difficult to discard when they were to be replaced by a standardized method. There was also the matter of bruised pride, a definite psychological side effect. However, experienced programmers soon became convinced of the validity of standardization, based on their past experience and the obvious benefits. For example, because of the standardized method of coding, code reading proved to be a valuable desk debugging tool.

Toward the end of phase I, it became evident that maintenance of programs was easier. As is mentioned in Section III, the maintenance activity for each test set peaked during the first two months following delivery. Maintenance requirements generated by debugging activities generally required one programmer full time for that period. Structuring techniques made the programs easily readable and enabled them to become community property. In fact, this standardization was so effective that, immediately following delivery, maintenance of all programs in a test set could be assigned to one member of the original developing team. Maintenance responsibility included an average of 100 programs per test set. Transferability of program maintenance thus had the effect of freeing key personnel for scheduled critical design activities for the next test set, as well as lessening the impact of loss of personnel through rotation. Orientation of new personnel was also simplified, since this could be partially accomplished through code reading.

VII. PROGRAM PRODUCTION LIBRARY AND THE LIBRARIAN

The Program Production Library (PPL) facilitates the work of programmers engaged in code development; it also aids project and line management wishing to review the project's progress. The PPL depends on a computerized library system in which all types of data have a defined source and destination. It is maintained by clerical personnel, but no operations are carried out in it unless they are directly requested by the programmers.

Program librarians staff the PPL. Just as structured programming must be introduced slowly, the program librarian must be given adequate time to learn. The librarian's first job is to provide an interface with the computer center, submitting and picking up jobs. The librarian can later be taught to change source code, working from marked-up program listings. The skills required for this are the ability to interpret the sequence of source changes, to make up the appropriate change deck, to incorporate this change deck in the necessary computer input deck so the program source change will be made, and to include the proper tests so that the programmer will have a new set of outputs to analyze. This represents a high level of proficiency for a program librarian, yet it requires no programming skills.

The librarian is also responsible for maintaining current listings for all programs being developed. During the development of interdependent programs, library listings must be updated daily, since several programmers may be working on the same program or require interface to a common data area. However, on the project studied, each test in the set was designed so that all required predecessor conditions were established during the test. Each team member was assigned a specific test, and, since only the programs within a test were interdependent, it was not necessary to file final listings until they were ready for debugging.

VIII. PHASE II

Phase II involved the introduction of the programming production library and the program librarian. The overall effects observed during phase II were not immediately visible. This was due mainly to the learning curve of the program librarians. The acceptance of the librarian service and the PPL concept was not universal, and it occurred much more slowly than the acceptance of structured programming techniques. Initially, it had the effect of placing one more barrier between the programmer and successful computer output. During the project studied, the training of new librarians was a continuing activity, because of frequent turnover. One month overlaps for training were worked into the schedule, increasing the overall manpower required to

support the PPL. During that training period, library performance usually suffered. This also hampered the expansion of PPL functions, since overall accuracy of PPL output varied. It took four to six months before programmers began to rely entirely on the librarian service.

The sporadic accuracy and reluctant acceptance of the PPL and librarians can be attributed almost entirely to frequent turnover of librarian personnel.

The librarian's job is not trivial and requires about two months of close supervision by trained personnel to achieve the basic skill of job setup using change decks provided by programmers. Six months after phase II began (Test Set 7, Table I), the impact of the training period had been mitigated by increased experience and improved PPL procedures that defined additional fail-safe measures for new personnel. For the remainder of the study, PPL throughput and accuracy increased despite continuing turnover.

Another effect of this turnover was the operation of the PPL on a "pool" basis. Since experienced librarians were scarce, all PPL activities were centralized into a pool of three to four librarians shared by four programming departments. This arrangement was quite effective in handling the peak activity periods that precede each delivery.

The number of librarians required for such a pool varies according to the amount of new development being done, the number of programmers involved, and their skill level. The ratio used in the environment described here was 6:1; that is, one librarian for every six programmers. These personnel were not added to the programming group. Instead, given the 6:1 ratio, in a group of seven programmers with an average experience level of two years, one programmer was replaced with one librarian. The remaining six programmers then produced the same amount of code with the aid of the librarian as the original seven programmers would have produced without the librarian.

Another observation is that personnel new to programming can gain programming experience quickly since they are not concerned with the detailed procedures required for job submission and job handling. They need only concentrate on the technical aspects of programming.

IX. CONCLUSION

Standardization of programming techniques through structured programming and its related practices leads to increased maintainability. Background maintenance activities are more easily rotated since structured programs become community property. The PPL concept extends standardization to the programmer/computer interface and as such is beneficial. The role of the program librarian removes as many clerical tasks as possible from programmers, allowing them to

concentrate more directly on the technical content of development. The productivity trend indicated in Table I is presented to indicate the effect of these new technologies on programmers. Obviously, productivity should increase as programmers are freed of time-consuming clerical tasks as indicated by phase II. However, it can also be seen that productivity in phase I increases simply with the use of standardized programming techniques.

REFERENCES

1. C. Bohm and G. Jocopini, "Flow Diagrams, Turing Machines," Communications of the ACM, *9*, No. 4 (May 1966).
2. J. P. Haggerty, "SAFEGUARD Data-Processing System: Central Logic and Control Operating System," B.S.T.J., this issue, pp. S89–S99.
3. J. W. Olson, "SAFEGUARD Data-Processing System: Architecture of the Central Logic and Control," B.S.T.J., this issue, pp. S41–S61.