SOFTWARE DEVELOPMENT: A SURVEY OF CURRENT PRACTICES

by

LOYE E. HENRIKSON

B. Sc. Ed., Central Missouri State University, 1974

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

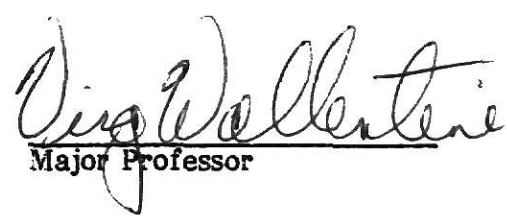
MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1982

Approved by:



Major Professor

TABLE OF CONTENTS

LIST OF TABLES

ACKNOWLEDGEMENTS

The author wishes to express his gratitude to the many people who helped him with this study.

His committee members, Dr. David Gustafson, Dr. William Hankley, and Dr. Myron Calhoun deserve praise for their guidance. Dr. Gustafson merits special thanks for his patience and understanding while serving as principal advisor.

Many thanks go to Larry Walker for shouldering much of the responsibility for the evening degree program in Kansas City and for keeping the author's motivation alive to complete the course work.

The respondents who took time to complete and return the survey deserve many thanks.

Finally, special thanks goes to his wife, Kathy, for her support and understanding and for her patience in typing, retyping, and proof-reading the report.

CHAPTER 1 INTRODUCTION

1.0 Software Development Costs and Techniques

A great deal of attention is paid these days to software costs, and understandably so. In 1976, overall software costs in the United States were estimated at more than $20 billion.[4] The continued decline in hardware costs and rise in software system costs naturally resulted in software accounting for an ever greater percentage of total system cost. This was seen as following the trend predicted by Boehm in 1973,[1] and would place software costs in 1976 at 80% of total. By 1985, if trends continue, software will account for 90% of total system cost.[1] This situation abounds with irony. Not only is hardware much cheaper, but it is also much more reliable than in past years, and the trend is toward still lower costs and even greater reliability in the future. Software, with its mushrooming cost, generally cannot match hardware reliability, and herein lies one of the ironies. Another is that, even as software becomes more expensive, it is not even generally recognized as a discrete, patentable entity. Anyone who has been involved in software development most likely thinks of the program or programs being developed as being very real, as of course they are. Anyone in management who has to budget software development costs has an equally good grasp of reality.

So, where does this leave us? We have cheap, reliable hardware and expensive, often unreliable software. Barry Boehm refers to builders of frequently unusable systems as "computer basket weavers". He states, "A basket weaver has a very difficult job. He must plan his basket very carefully and he puts a lot of loving care into it; he

builds it, studies it from various angles, discusses it with other basket weavers, and then goes off to build another basket. Very rarely does he go out and sample users to find out whether they are interested in baskets with handles or with several compartments rather than one compartment, and the like. And, unless something changes considerably in computing, it will remain a kind of computer basket weaving."[2]

In an effort to hold the line on costs and improve reliability (and productivity and quality), a great many software development techniques have been developed and used over the past few years for every phase of the software life cycle. A (nonexhaustive) list of recently developed software development techniques would include: software engineering, development of system requirements, software design methodologies, chief programmer teams, language preprocessors and macro generators, and automated software testing and evaluation systems. This report will review recently developed techniques and investigate the extent of their usage.

1.1 Software Engineering.

Software Engineering, considered soberly, should rightly include the entire software development process. Refining software development to an engineering discipline is in fact the direction software developers should take over the next few years. The present, unfortunate state is probably much closer to "basket weaving". Indeed, given the present conditions, "software engineering" practically amounts to a contradiction of terms. Hoare's brief, brilliant paper, "The Engineering of Software: A Startling

Contradiction"[11] states elegantly how far removed software development realities are from a literal application of software engineering. Witness: the engineer understands the needs of his client; the programmer "wishes he'd make up his mind what he wants". The engineer recommends from a range of known techniques those which are best for his client to minimize cost while achieving the desired effect; the programmer welcomes the most elaborate fancies of his "client" (user) as a challenge to his programming ingenuity. Still worse, the programmer often prefers to ignore known techniques used successfully by others, and embarks "on some spatchcocked implementation of (his) own defective invention".[11] How many of us have observed exactly this? How many of us are ourselves guilty?

Software Engineering, as an engineering discipline, should seek to reduce costs and improve reliability at every opportunity. Simplicity of every aspect from conceptualization to implementation is the only way to achieve these objectives. In large projects (100,000 source statements), design errors greatly outnumber coding errors both in sheer numbers and in effort required to detect and correct them.[3] One (as yet unresolved) difference between engineers and programmers is a widely accepted mathematical or theoretical foundation for programmers' work. The term "widely accepted" is key; a great many theoretical foundations for programming are available, but most are ignored. This would doubtlessly change if programmers, like engineers, were required to pass a licensing exam.

1.2 System Requirements.

Boehm estimates that 45-50% of software effort goes to the checkout and testing phase, and 30-39% to analysis and design.[1] Further, a generally undisciplined approach is usually taken in the analysis and design phase. One method of automated development of system requirements is described by Bell, Bixler, and Dyer. Their method is "Computer Aided Software Requirements Engineering", called SREM.[5] The system they describe would be most applicable to very large systems, and includes "techniques and procedures for requirements decomposition and for managing the requirements development process." It also includes a machine-processable language for stating requirements, and an integrated set of tools to support the development of requirements. The system is intended to add a measure of computer-imposed discipline to the requirements phase, an element almost invariably lacking. Of course, SREM is intended for use in designing truly large systems, such as the Ballistic Missile Defense system, which eventually had 8248 requirement paragraphs in a 2500-page specification.[5]

1.3 Software Design Methodologies.

Peters and Tripp have examined several software design methodologies, including: (1) Structured Design, (2) The Jackson Methodology, (3) Logical Construction of Programs, (4) META Stepwise Refinement, and (5) Higher Order Software.[16] The author of each of the above naturally has a different perspective of important design issues. Thus, "Structured Design" is concerned with data flow and its transformations from input to becoming output; the "Jackson Methodology" and the "Logical

Construction of Programs" hold that the identification of the inherent data structure is very important, and that the structure of the data (in and out) can be used to derive the structure of the program. Those who use "META Stepwise Refinement" state that success is certain if the problem is solved several times, each time being more detailed than before. "Higher Order Software" provides a set of axioms which must be used for successful software design.

Structured Design is based on concepts developed by Larry Constantine.[20] It relies primarily on following the flow of data through the system. Data transformations, transforming processes, and the order of their occurrences are depicted with a special notational scheme. The system specification is used to produce a data flow diagram, the diagram to develop a structure chart, and the structure chart to develop data structures. The use of structured design does seem to aid in rapid definition and refinement of data flows, but consistent identification of data transformations is sometimes difficult.[16] Structured design should work fairly well in systems containing only transformations that change data characteristics incrementally.

The Jackson Methodology sees data structure as the driving force of software design.[12] Programs are viewed as the means of transforming input data into output data. It is assumed that paralleling the structure of the input (data) and output (reports) will ensure quality design. Some cautions are needed, however: resulting data structures need to be compatible with rational program structure and only serial files may be involved. Data structuring is dependent upon the data base management system employed. Peters and Tripp object to the basic assumption of the Jackson Methodology: there is no simple linkage between data structures and program quality.[16]

Logical Construction of Programs (LCP) also assumes data structure as the key to software design, but is more procedural than the Jackson Methodology.[16] Originated by Jean-Dominique Warnier in France, some of it's methods have recently come into some use under the name of Warnier-Orr diagrams. The LCP method is:

1. Organize input data hierarchically (files, records, items).
2. Define the number of occurrences of each input element.
3. Do (1) and (2) for the output.
4. Obtain program details by identifying instruction types needed in a specific order: reads, branches, calculations, outputs, and subroutines.
5. In flowchart-like fashion, write the logical instruction sequence.
6. Number the parts of the logical sequence and expand each by using the instructions in step 4.

Some objections are that this method imposes hierarchical data structures inappropriately; for those cases in which hierarchical structures are appropriate, a pseudocode statement of a program is achieved rapidly, but may not be the program method normally chosen.[16]

META Stepwise Refinement[13] (MSR) involves beginning with a simple, general solution and building increasing amounts of detail until the final, detailed solution is reached. It requires a fixed problem definition, uses design levels, details are postponed to lower levels, the design is successively refined, and correctness is assumed at each level. MSR was designed by Henry Ledgard, and is a blend of Mill's top-down ideas, Wirth's step-wise refinement, and Dijkstra's level structuring. MSR attempts to separate functionally independent levels: higher levels being the general problem statement and lower ones being more detailed. Modules at a certain level may invoke only lower-level modules.

MSR's theory is good, but in reality, non-trivial problems undergo constant revision. Since MSR's solutions at any given level are dependent upon prior levels, and any change in the problem affects prior levels, the ability to produce a solution at any given level is lacking unless all levels are up to date.[16] MSR perhaps works best on a small, well-defined problem requiring an elegant solution, such as an executive for an operating system.

Higher Order Software (HOS) was developed by Hamilton and Zeldin while working on NASA projects at MIT. It was intended to provide a formal means of defining reliable, large-scale, multiprocessor systems.[10] The main parts of HOS are:

1. a set of formal laws,
2. a specification language,
3. automated analysis of system interfaces,
4. system architecture layers from analyzer output, and
5. transparant hardware.

The design method is based upon axioms defining a hierarchy of software control, with control being a formally specified effect of one software module upon another. The axioms, which are very explicitly stated, define intermodular controls, access rights to variables, and relationships between modular levels.

HOS is most useful in applications wherein the accuracy and auditability of the algorithm are the primary concerns, such as in scientific problems. Peters and Tripp object to HOS' lack of explicit data base design concepts. Their experience with large systems is that design of code and data base must be synchronous.[16]

In comparing these five methodologies, Peters and Tripp conclude that no single method applies to every situation, that designers produce designs, methods do not, and finally, that designing is problem solving, a personal issue. Methods are most successfully applied in supportive management environments which include planning, scheduling, and control. Merging methods and environments would perhaps be the next step.[16]

Of the five methodologies above, Structured Design is perhaps the most used, or at least most often mentioned. Quite possibly references to "structured design" are not to the developments of Constantine,[20] but instead to general techniques. Indeed, structured design has become something of a generic term, admittedly preferable to "basket weaving". In spite of their limitations, the five methodologies represent important steps in organizing the design phase of the software life cycle. It would be interesting to investigate the extent of usage of these methodologies, perhaps by making reference not to particular methods by name but by less specific generic references. This would allow respondents to answer questions concerning usage of structured techniques without requiring familiarity with specific literature.

1.4 Chief Programmer Teams.

Many production projects are staffed by relatively junior programmers. The low average level of education and experience usually results in less-than-optimum efficiency in program design, coding, and testing. More experienced programmers, who probably have the needed insight and knowledge necessary to improve this situation, are usually in low-level management positions where they cannot

accomplish the detailed programming work.[6] In this typical project structure, each programmer normally has complete responsibility for all aspects of one or a few modules. This entails not only programming, but also maintaining listings, setting up runs, and writing reports concerning all aspects of his portion of the system. Few guidelines (or standards) for any of the associated tasks are ever provided, so the results are very individualized. This often further complicates the process of system integration, documentation, and leads to a lack of development effectiveness.

One approach to attacking the problems of poor software quality and low programmer productivity is the chief programmer team. If, indeed, programming is the most complex mental activity ever undertaken by mankind,[7] then such an approach should be elementally appealing.

Baker [6] describes a chief programmer team management concept for production programming. He describes a chief programmer as a senior-level programmer who is given overall responsibility for development of a production system. The chief programmer produces the nucleus of the system being developed, and specifies and integrates other programming activity for the system. Other permanent team members are his backup programmer (also senior level) and a programming librarian. The team is described as analogous to a surgical team, with the chief programmer being compared to the chief surgeon, and being supported by a team of specialists who assist the chief. Functional details of the system may be provided by other programmers and then integrated into the system by the chief programmer. This approach directly attacks many of the problems of the more traditional approach, particularly coordination of development.

1.5 *Language Issues.*

Programming languages are perhaps often overlooked as potential tools for improving program reliability. Program reliability may be considered in two aspects: (1) Number of errors which crop up after system release, and (2) correct, predictable performance of intended function. The second is the more important of the two, although both are important. Reliability features should include robustness (the ability of a system to handle unexpected data values gracefully), and a high degree of system availability (infrequent crashes).

Language features important for reliability are:

Simplicity - *Language must be small enough for programmers to master.*

Consistency - Language should form coherent whole - regular semantics, uniform syntax.

Modularity - Language must support the division of a problem into smaller problems.

Redundancy and error checking - with good diagnostics.

It seems fairly obvious that programs with a high probability of being correct are more reliable. Two methods of increasing this probability are: reducing the probability of programming errors and increasing the probability of detecting errors.[8] Good language design can contribute to both goals. A programming language should lend confidence to the programmer in the correctness of programs. Violations of programmer intentions should be detectable as errors, ideally at compile

time. Redundant expression of intention through type checking, declarative redundancy, and assertions should lead to better error detection.[8] Other common sense issues are really language independent. Eliminating "magic numbers" outside of named constants and using readable formatting of program text on the page are two activities which may be accomplished with a little programmer effort in almost any language. A nicer approach would be the automation of these in the design of a language.

Language improvements need not be limited to cosmetic improvements, of course. Gannon and Horning mention statistically significant results demonstrating that subjects using a nestable conditional construct made fewer semantic errors and arrived at problem solutions more quickly than did subjects using a branch-to-label construct.[8] This may be a matter of programmer training and technique, but it also could be a built-in language construct (or omission of one in the case of branch-to-label). Potentially, programming languages could accomplish much in improving program quality and reliability.

The language used by more software developers than any other is COBOL. Philippakis surveyed 164 computer installations and found that 86% used it, and that 70% of coding was done in COBOL.[17] Unfortunately, COBOL is frequently attacked from many standpoints, including intrinsic design and error-proneness.

To consider an example of intrinsic COBOL design limitations, observe some code which might be generated by a programmer attempting to use structured programming techniques:

```
READ-AND-PROCESS-DATA.
    READ DATA-FILE
        AT END MOVE 1 TO END-OF-FILE-FLAG.
    PERFORM PROCESS-DATA UNTIL END-OF-FILE.
    .
    .
    .
PROCESS-DATA.
    Process data record
    WRITE DATA-RECORD.
    READ DATA-FILE
        AT END MOVE 1 TO END-OF-FILE-FLAG.
PROCESS-DATA-EXIT.
```

Small wonder that many programmers find structured programming awkward. Some rather convoluted logic is involved in coding a "pre-read" before entering the loop which really does the major portion of the work, the PROCESS-DATA paragraph. Further, PROCESS-DATA is itself of an awkward structure. At first glance, the logic seems all wrong: you should read first, then process and write data. If PROCESS-DATA were physically far removed from it's calling sentence (as it often would be), the observer would usually want to look at the calling sentence to assure that the logic in fact is correct.

Now, consider an imaginary extended COBOL version:

```
LOOP.
    READ DATA-FILE
        AT END EXIT LOOP.
    Process data
    WRITE DATA-RECORD.
ENDLOOP.
```

This example allows utilization of the read-write sequence, which seems more natural than write-and-read (with a pre-read first). The problem is that standard COBOL has no in-line-loop capability, nor a general EXIT-IF structure. A language preprocessor would, of course, allow just this sort of structure, and many such preprocessors are available today, MetaCOBOL being one example.[21] Another approach would be the use of a macro facility. The macros themselves would be written in COBOL and configured with the COBOL compiler. The macros would be converted to standard COBOL statements at compile time. Perhaps some future ANSI version of COBOL will include in-line looping, exit-if, and other such constructs.

Error-proneness is another COBOL vulnerability. Research conducted by Litecky and Davis indicated that 20% of error types account for 80% of the total error frequency in COBOL programming.[14] Somewhat surprisingly, the more persistent errors were clerical mistakes: adding a period after 'FD file-name', the use of commas as word delimiters, etc. These errors do not involve crucial elements of the basic structure of COBOL, which strongly suggests that such elements should be changed. Another finding of their research was that 80% of error diagnostics were inaccurate, a problem which should also be improvable, particularly if emphasis were placed on better diagnosis of common errors.

1.6 Automated Software Testing and Evaluation.

Upon completion of coding, testing and debugging begins. This can be the most tiring, expensive, and unpredictable phase of software development, and often represents 40-50% of the total effort.[1] Large systems may consist of many components with complex interactions, probably developed by a large number of programmers. For these reasons, operational software is often not error-free in spite of large testing efforts.

Formally proving the correctness of large, complex systems is not currently feasible, although proofs of non-trivial programs are possible.[9] Ramamoorthy and Ho describe automated tools for all phases of software development.[19] Automated tools are programs to check such things as program syntax, control structures, module interface, and testing completeness. These tools may be used to allow programmers to concentrate on advance system checkout by removing simple coding errors. Among those for the testing phase are tools for monitoring program run-time behavior and automated test case generation. Monitoring run-time behavior might include bounds checking, recording frequency of traversal in particular sections of code, and excution path tracing, which would record paths taken by test cases. Test cases may be generated to exercise all possible program branches, sometimes considered "complete" testing. Exhaustive testing is usually impossible, so "complete" testing is defined in a more relaxed manner. It is possible to find a minimum set of test cases to accomplish this, and Ramamoorthy and Ho describe an algorithm to detect branches not tested and indicate conditions necessary to traverse those paths.[19]

Problems encountered in developing software systems are usually tackled in an ad hoc manner. Although most software errors are design errors,[3] most software evaluation systems attempt to solve problems at the code level, which is well after design is complete. This condition could be corrected by designing an automated evaluation system in conjunction with the development methodology. Systems should be designed with the goal of validation in mind.[19]

1.7 Application of New Techniques.

As may be surmised from the preceeding sections, many software development techniques have been developed over the past years. Most are designed as tools for use in very large projects such as missle systems, space programs, or perhaps compiler development. Their use should enhance management control over development cost. This goal is imperative, given the well-established upward trend of software development cost.

These newly developed design techniques should be equally applicable to smaller projects. Few businesses undertake projects of the same scale as a missle system, but are nontheless interested in controlling costs. The use of a programming language is elemental in any software development project. It is thus an interesting area of focus in an assessment of usage of new design techniques.

In an effort to do this, a questionnaire was sent to 58 data processing shops. Emphasis was placed on questions concerning programming languages. The

geographical area covered was limited to the Kansas City area in the interest of easier communication with respondents and in time constraints. It is acknowledged that Kansas City may not be representative of general data processing practices, but hopefully the survey results may serve as a basis for further investigation.

CHAPTER 2 QUESTIONS ON THE SURVEY

Ideally, a survey should be constructed to achieve a good response rate but also provide an adequate volume of usable data. The greatest difficulty in writing a survey lies in satisfying these two conflicting goals. The rate of response should be enhanced by writing a limited number of easily answered questions using generally understood terminology. This is no small task, particularly if information is desired concerning recently developed techniques. Usable responses are those which are readily quantifiable and yet reflect reality at the respondent's shop. Anticipating common responses and including these as choices in multiple-choice questions is one step that may be used in gathering standard data.

These factors were considered in writing the survey of software development practices. Top priority was given to achieving a good response rate. To limit the number of questions, emphasis was placed on language usage; a total of five language-oriented questions were included. Only one question was asked concerning the extent of usage of new design methodologies. One question was also asked about software development problems. Guaging language usage and satisfaction in applications development thus became the main thrust of the survey.

The questions and some comments follow.

1. Title of person responding

   - was requested to determine if responses would differ between management and staff personnel.

2. Primary programming language used in shop.

3. Are most applications business or numerically oriented?

   - was asked to see if software development practices would differ between business and scientific or engineering shops.

4. Are you happy with the language?

   - It was anticipated that a certain amount of dissatisfaction would be expressed, particularly in shops experiencing problems.

5. Are most applications programmers happy with the language?

   - It was expected that even greater dissatisfaction would be expressed than from management.

6. Have you considered converting old applications or writing new ones with a different language?

   - This was asked to lend further qualification to the responses from questions (4) and (5). It was thought also that trends toward usage of new languages could be spotted.

7. Would you like to see improvements in your language? Examples:

   a. CASE statement

   b. DO...UNTIL or WHILE cond. DO...

   c. Cross reference of called procedures

   d. Local data in procedures

7. (cont'd)

   e. More accurate error diagnostics

   f. Other

As in question (6), it was thought that responses to this question might help clarify the question of language satisfaction, although indicating a desire for improvements would not necessarily imply dissatisfaction.

Examples (a), (b), and (d) are not available in more commonly used languages (COBOL and FORTRAN), although these three may be simulated by the use of preprocessors. They were chosen as examples because of their importance in structured programming and because of their high frequency of simulation by using existing language constructs. The other two, (c) and (e), are frequently mentioned as desirable complier enhancements.

8. Does your shop utilize structured techniques? Examples:

   a. Top-down design

   b. Step-wise refinement

   c. Hierarchic control structures (perform, gosub, etc.)

   d. Modular programs/segments

   e. Extensive use of comments

   f. Other

Both design and programming techniques were included in the chosen examples; these probably should have been separated into separate questions. Top-down design is a commonly mentioned structured design technique; step-wise

refinement is less commonly known. Section three in Chapter 1 contains details about Structured Design and META Step-Wise Refinement; examples (a) and (b) referred to these. The remaining choices (c), (d), and (e) are structured programming techniques.

9. What are the most serious software development/maintenance problems in your shop?

   a. Cost/time overruns
   b. Poor communications with users
   c. Personnel turnover
   d. Unreliable hardware
   e. Inadequate development tools
   f. Lack of education or experience among staff members
   g. Other

   Decreasing the frequency of cost/time overruns was one of the more important reasons for development of techniques described in Chapter 1; it was thought that it would be interesting to see if the use of any particular structured design techniques were effective against overruns. The other examples were chosen as commonly occurring problems in software development; all may be contributors to cost/time overruns. Asking respondents to rank the seriousness of problems in their own shops might have been a better approach.

It was expected that a majority of shops would use COBOL, so the language questions were perhaps biased toward responses from COBOL users. For example, it was expected that many COBOL users would express dissatisfaction with their language.

The results were surprising: 94% of managers were satisfied, and indicated that 100% of their programmers were also. (This was for all languages, not just for COBOL.) A hint of this bias may also be evident in the next two questions concerning converting to a new language and inquiring about language improvements desired. It was anticipated that some interesting remarks might come from COBOL users responding to these questions. This proved to be not significantly so.

Consideration was given to including questions about previous language usage and software development practices. These were omitted from the survey to remain consistent with the goal of limiting the number of questions asked in hope of achieving a high rate of response. Later, during analysis of survey responses, it was observed that historical data would have been very useful in spotting trends, particularly in the occurrence of software development problems. (This idea is addressed in more detail in Chapter 5.) It was believed that useful information concerning current practices could be obtained with the chosen questions and that in the worst case, a benchmark could be established for later software development surveys.

CHAPTER 3 OVERALL SURVEY RESULTS

3.0 Response Profile.

Of 58 questionnaires sent to Kansas City area locations, 37 were returned, giving a 64% response rate. Some effort was made to include a broad range of industries, as well as shop sizes, although no research was done to identify businesses within specific budget ranges or numbers of employees. With this limitation in mind, the sample is generally representative of the Kansas City area. Industries and number of respondents were:

Education	4
Engineering Consultant	2
Health Care	3
Insurance	3
Manufacturing	10
Retail/Distribution	5
Utility	4
Other	6
Total	37

Only two non-management responses and two non-business shops were received, so no comparisons using these were possible.

Although efforts were made to write questions which would generate quantifiable responses, it became evident during response analysis that some improvement was

possible. For example, question 9 asked which of 6 possible software development problems were most serious in the respondent's shop. This allows serious problems to be identified, but provides no measure of the severity. A better method of stating this question might have been to ask the respondent to indicate which problems were encountered and to rank these according to seriousness. In evaluating the results of this survey, it is of course possible to note such things as the total number of occurrences of a particular problem, say cost/time overruns, but it is not possible to determine the overall "seriousness ranking" of any particular problem. With this limitation in mind, the results probably reflect reality as perceived by the respondents.

3.1 Preliminary Observations.

The following conclusions may be drawn from Table 3.1-1:

1. COBOL is the most used language, listed by 84% of respondents. This result corresponds very closely to the 86% usage found by Philippakis in 1974.[17] COBOL is apparantly not being replaced by other languages.

2. It was possible to identify eight shops (22% of those responding) using assembler. Twenty-two percent is a much lower usage than the 76% found by Philippakis,[17] but it must be emphasized that the 22% is a very low-confidence figure. Six of the eight were identified by their indication that they were converting old assembler programs to COBOL. It is possible, even likely, that many more shops use assembler than were identified by

the survey. It would appear, however, that fewer shops use assembler than in 1973 when the Philippakis survey was published, and that those using assembler are tending toward higher level languages.

3. Language satisfaction is very high among both management and programming staff.

4. COBOL is the majority choice of those considering a different language for new applications.

5. The use of structured design and programming techniques is widespread.

6. Even more widespread is the occurrence of software development problems.

Another observation from the data in Table 3.1-1 is that slightly more than half of the respondents would like language improvements. A positive correlation was found between the use of COBOL and the desire for language improvements, but the correlation was of moderate statistical significance, between the 1 and 5% levels.

Table 3.1-2 shows language improvements desired in some detail. Percentages shown in the table are not of the entire sample, but of those desiring improvements. The following conclusions may be drawn:

1. More accurate error diagnostics and a cross reference of CALL'ed procedures are the most often listed improvements. Both are really compiler enhancements, not language improvements.

2. In-line looping and 'CASE' statements are of about equal desirability, but both are fairly low.

The percentages of those desiring more accurate error diagnostics would seem to show increasing demand with increasing shop size, but no correlation of statistical significance was found to support this idea. More generally, no significant correlations were found between shop size and any of the language improvements.

The use of structured techniques is documented by Table 3.1-3. It may be noted that the percentages of shops using top-down design increases with shop size. A positive correlation was found to support this observation, but was below the 5% level of significance. A very similar statement may be made for the use of comments: use increases with shop size, a positive correlation may be found, but is of marginal significance. Included under the category of "other" are three instances of the use of Warnier-Orr diagrams.

The occurrence of software development problems is tabulated in Table 3.1-4. The only correlation between shop size and the occurrence of any particular development problem is that between size and cost/time overruns. A positive correlation was found, significant to the 5% level. Lack of education or experience is easily the most frequent problem, and appears to decrease with shop size. However, no significant correlation was found in this relationship.

Table 3.1-1 - Summary of Results

CATEGORY	SHOP SIZE							
	15	41%	15	41%	7	18%	37	100%
	SMALL		MEDIUM		LARGE		ALL	
	No.	%	No.	%	No.	%	No.	%
1. Primary Language(s):								
COBOL	12	80%	13	87%	6	86%	31	84%
Assembler Languages	1	7%	1	7%	1	14%	3	8%
FORTRAN	1	7%	1	7%	0	0%	2	5%
RPG II	2	13%	1	7%	0	0%	2	5%
MARK IV	0	0%	2	13%	0	0%	2	5%
MIIS	0	0%	1	7%	0	0%	1	3%
2. Application Type:								
Business	14	93%	14	93%	7	100%	35	95%
Business/Numeric	1	7%	1	7%	0	0%	2	5%
3. Management Satis. w/Lang.	15	100%	13	87%	7	100%	35	95%
4. Prog. Satis. w/Lang.	15	100%	15	100%	7	100%	37	100%
5. Consider Different Lang.	4	27%	8	53%	3	43%	15	41%
6. Different Lang. Considered:								
COBOL	3	75%	5	63%	2	67%	10	67%*
Other	1	25%	3	38%	1	33%	5	33%*
7. Would Like Lang. Improvs.	8	53%	9	60%	4	57%	21	57%
8. Use Structured Techniques	11	73%	11	73%	5	71%	27	73%
9. Experience S.W. Dev. Probs.	10	67%	15	100%	6	86%	31	84%

*Percentage of those considering a new language

Table 3.1-2 - Language Improvements Desired

Language Improvements Desired	SHOP SIZE							
	8	53%	9	60%	4	57%	21	57%
	SMALL		MEDIUM		LARGE		ALL	
	No.	%	No.	%	No.	%	No.	%
CASE	3	38%	2	22%	2	50%	7	33%
DO...UNTIL or WHILE...DO	2	25%	4	44%	0	0%	6	29%
'CALL' cross reference	2	25%	6	67%	2	50%	10	48%
Local Data	2	25%	1	11%	1	25%	4	19%
More accurate diagnostics	4	50%	6	67%	3	75%	13	62%
Other	2	25%	4	44%	1	25%	7	33%

Table 3.1-3 - Use of Structured Techniques

Structured Design Techniques	SHOP SIZE							
	11	73%	11	73%	5	71%	27	73%
	SMALL		MEDIUM		LARGE		ALL	
	No.	%	No.	%	No.	%	No.	%
Top-down Design	4	36%	7	64%	4	80%	15	56%
Step-wise refinement	1	9%	1	9%	0	0%	2	7%
Hierarchic control structures	9	82%	6	55%	4	80%	19	70%
Modular programs/segments	8	73%	5	45%	4	80%	17	63%
Comments	4	36%	6	55%	4	80%	14	52%
Other	2	18%	5	56%	2	40%	9	33%

Table 3.1-4 - Software Development Problems

Software Development Problems	SHOP SIZE							
	10	67%	15	100%	6	86%	31	84%
	SMALL		MEDIUM		LARGE		ALL	
	No.	%	No.	%	No.	%	No.	%
Cost/time overruns	3	30%	5	33%	5	83%	13	42%
Poor communications w/users	1	10%	6	40%	3	50%	10	32%
Personnel turnover	4	40%	6	40%	0	0%	10	32%
Unreliable hardware	1	10%	2	13%	1	17%	4	13%
Poor development tools	2	20%	1	7%	1	17%	4	13%
Education/Experience	7	70%	9	60%	1	17%	17	55%
Other	1	10%	3	20%	1	17%	5	16%

CHAPTER 4 ANALYSIS OF DATA

4.0 Investigations for Relationships.

Potentially, one of the more interesting findings of a survey of this nature would be the existence of software development techniques effective in controlling development problems. No such conclusions appear possible from analysis of data collected by this survey. Possible relationships between occurence of software development problems and desire for language improvements were also investigated, but the results were largely inconclusive.

4.1 Analysis Techniques.

Analysis techniques used were:

1. A table of the occurences of specific software development problems in conjunction with the usage of specific structured techniques was drawn. (See Table 4.2-1).

2. The same table as in (1) was drawn, but showing instead, percentages of those using structured techniques and not using structured techniques. (See Table 4.2-1a).

3. A table was drawn showing the total number of software development problems against the total number of structured techniques used for each

respondent. Possible correlations were investigated between the number of problems and the number of techniques. (See Appendix B).

4. Software development problems and structured techniques were ranked respectively by perceived seriousness and perceived value or potential effectiveness. Total problem and technique "scores" were then computed, placed in a table, and then investigated for correlation. (See Appendix C).

5. A table was drawn as in (3) above, but showing the total number of software problems against the total number of language improvements desired by each respondent. Possible correlations were then investigated. (See Appendix D).

6. A table similar to (1) above was drawn, showing the occurence of specific software development problems in conjunction with language improvements desired. (See Table 4.2-2).

7. Another table similar to (3) above was drawn, but this time showing only the number of structured programming techniques used against the number of language improvements desired. Correlations were investigated. (See Appendix E).

8. Additional correlations were investigated and are detailed in Appendix F.

4.2 Analysis Results.

The results of the above analysis techniques, taken in the same order:

1. No quantitative correlation calculations were possible due to the unordered nature of the tabular data. In terms of raw numbers, it would appear that more shops using structured techniques experience software development problems than do those using no structured techniques.

2. No quantitative correlations were possible for the same reason as (1) above. The conclusion drawn in (1) no longer holds, however. It is not possible to conclude that greater percentages of shops using structured techniques experience software development problems than do those using no structured techniques.

3. A positive correlation just outside the 5% significance level was found between total numbers of software design problems and total numbers of structured techniques used. (See Appendix B).

4. Problems were ranked in order of ascending perceived seriousness: (1) unreliable hardware; (2) inadequate development tools; (3) lack of education or experience; (4) poor communications with users; (5) personnel turnover; (6) cost/time overruns. The individual respondent scores, when added, ranged from zero to 21. The ranking scheme was chosen partially by the total number of occurrences of each problem, particularly in the

case of unreliable hardware and inadequate development tools. Cost/time overruns was considered the most serious problem, even though it was not the one most frequently occurring.

Structured techniques were ranked by supposed effectiveness, also in ascending order: (1) use of comments; (2) modular programs; (3) hierarchical control structures; (4) step-wise refinement; and (4) top-down design. Step-wise refinement and top-down design were assumed equally effective, and, if both were used by a respondent, only one was counted. Individual respondent scores ranged from zero to 12.

Using the ranking technique just described, a small positive correlation was found between the occurence of software development problems and the use of structured techniques. The correlation was slightly less than that found in (3) above, and so was also outside the 5% significance level. (See Appendix C).

5. A strong positive correlation (inside the 1% significance level) was found between the occurence of software development problems and the desire for language improvements. (See Appendix D).

6. As in (1) above, no quantitative correlations were possible because of the unstructured nature of the data. It would appear, however, that more respondents experiencing software development problems were in favor of language improvements than those who were not.

7. Two separate cases were considered: (a) use of structured programming techniques (not including structured design) against language improvements desired (not including compiler improvements) and (b) use again of structured programming techniques only against language and compiler improvements. Case (a) showed a positive corrleation just below the 5% significance level. Case (b) showed no correlation. (See Appendix E).

8. Positive correlations inside the 5% significance level were found between (1) the use of COBOL and the desire for language improvements and (2) shop size and cost/time overruns. (See Appendix F, Tables F-I and F-IX). No other significant correlations were noted. (See Appendix F).

Table 4.2-1 - Development Problems vs. Structured Techniques

		SOFTWARE DEVELOPMENT PROBLEM							
		Ap	Bp	Cp	Dp	Ep	Fp	none	
	At	6	5	7	3	1	9	0	At = top-down design
	Bt	0	0	1	0	0	1	0	Bt = step-wise refinement
STRUCTURED	Ct	5	3	6	2	1	7	1	Ct = hierarch. contr. struc.
TECHNIQUES	Dt	7	4	8	3	2	9	1	Dt = modular programs
	Et	5	3	4	1	0	7	0	Et = use of comments
	none	2	2	1	0	2	3	1	

Ap = cost/time overruns
Bp = poor communications with users
Cp = personnel turnover
Dp = unreliable hardware
Ep = inadequate development tools
Fp = lack of education or experience

Table 4.2-1a - Development Problems vs. Structured Techniques
(In percentages of those using (or not using) structured techniques.)

		SOFTWARE DEVELOPMENT PROBLEM						
		Ap	Bp	Cp	Dp	Ep	Fp	none
	At	30	22	26	15	4	33	0
	Bt	4	0	4	0	0	4	0
STRUCTURED	Ct	26	19	22	11	4	26	4
TECHNIQUES	Dt	33	15	30	15	7	33	4
	Et	26	15	19	7	0	26	0
	none	20*	20*	20*	0*	20*	40*	10*

*Percentage of those not using structured techniques

Table 4.2-2 - Development Problems vs. Language Improvements

		SOFTWARE DEVELOPMENT PROBLEM							
		Ap	Bp	Cp	Dp	Ep	Fp	none	
	Ai	4	3	5	2	2	4	0	Ai = CASE statement
LANGUAGE	Bi	2	4	4	1	1	6	0	Bi = DO until; WHILE do
IMPROVEMENT	Ci	5	6	6	4	2	5	1	Ci = 'CALL' cross-ref.
DESIRED	Di	2	1	5	3	3	5	0	Di = Local data
	Ei	6	6	4	3	1	5	1	Ei = better diagnostics
	none	4	2	3	0	2	5	5	

Ap = cost/time overruns
Bp = poor communications with users
Cp = personnel turnover
Dp = unreliable hardware
Ep = inadequate development tools
Fp = lack of education or experience

CHAPTER 5 CONCLUSIONS

5.0 Overall Results.

Certain observations may be made concerning overall results:

1. COBOL is probably as extensively used as in 1974.
2. Structured techniques are in wide use.
3. Software development problems are still widespread.

An unavoidable conclusion from the analysis just completed in Chapter 4 is that more respondents using structured techniques experience software development problems than do those who do not. There are at least four possible explanations:

1. This conclusion is accurate.
2. Shops not using structured techniques and reporting no software development problems fail to realize that problems really are present.
3. Shops experiencing software development problems are using structured techniques to respond to the problems, which may or may not be improving.
4. The questionnaire contained too few questions and the sample taken was too limited to support or deny this conclusion.

If the results are assumed accurate (1), then perhaps the recently developed software design techniques are ineffective. The structured techniques referred to in the questionnaire include both structured design and structured programming, but not

individually, and this is itself, incidentally, a weakness of the survey. But to consider just structured programming for a moment, structured programming (SP) is sometimes thought of as a discipline for turning out programs in response to assignments. Certainly SP as covered by the questions and responses to the questionnaire fits that description. Prywes states that the improvements in the programming process due to the current methodology of SP are inadequate to respond to future needs.[18] Perhaps the results of this survey are evidence in support of this conclusion.

Another possibility is that outlined in explanation (2). Many shops utilizing traditional software development techniques (junior personnel, individualized responsibility, few guidelines, etc.; see Chapter 1, Section 1.4 for more detail) may experience poor efficiency in program design, coding and testing. The state of software development may be very poor but accepted as normal by management, which may not realize that things could be much better. If there were little concern about productivity or reliability, the result would be much the same: serious problems could exist but would be unacknowledged by management.

There is no method of verifying or disclaiming explanation (3) by using data from the questionnaire. For reasons already stated (limiting the survey length), no questions were included regarding historic practices or problems. This line of reasoning leads directly to consideration of explanation (4), that the questionnaire was too short, and the sample taken was too limited.

5.1 Suggestions on Improving the Survey.

Some weaknesses of the questionnaire have already been stated, namely problems with the wording of certain questions and the lack of questions regarding previous practices. It should be possible to construct a questionnaire correcting both of these difficulties, and perhaps adding a few other features as well. It would seem desirable to ask questions about the use of structured design techniques separately from structured programming techniques. The question about language usage would be improved by including a list of often used languages and asking the respondent to indicate the percent of current and recently past coding effort experienced by each language.

A good approach to structuring the sequence of questions might be to parallel the software development cycle itself. The order would consist of questions regarding the use of recently developed approaches to: (1) development of system requirements, (2) software design methodologies, (3) software engineering, (4) language usage, and, (5) automated software testing and evaluation systems. The questions should request both current (within the past 12 months) and recent past (perhaps 13-36 months) information to determine trends and the degree of effectiveness of particular practices. The questions should be constructed so that the responses would be more readily quantifiable. An example might be:

> Indicate on the scale below your current use of top-down system design with an "N" and your use over the past 13-36 months with a "P":
>
> None Moderate Extensive
>
> 

A set of this type of question could then be subjected to a Likert-type analysis.[22]

Another approach might be to construct most or all of the questions in the same manner as stated above for the language usage question. Commonly used (or recently developed) structured programming techniques, for example, could be listed and the respondent asked to estimate the percent of current and past programming effort expended in utilizing each technique. This approach should allow the computation of correlations between the use of certain techniques and the occurence or discontinued occurence of problems. It should allow trends to be spotted and conclusions to be drawn concerning the usage and effectiveness of structured design and programming techniques.

The sample taken should not be limited to any particular geographic area. The target population should be selected with some care to get a cross-section of large, medium, and small businesses with respect to both budgeted amounts and staff size. Effort should be made to sample a variety of industries to obtain an overall representation of current practices.

Many of the results of this survey are inconclusive, but hopefully the report may serve as a basis for further investigation. Much work remains to be done before any strong conclusions may be drawn concerning the use and effectiveness of structured design and programming techniques. Designing an improved questionnaire would be the first step.

Bibliography

1. Barry W. Boehm, "Software and Its Impact: A Quantitative Assessment", Datamation, Vol. 19, No. 5, May 1973, pp. 48-59.

2. Barry W. Boehm, "Command/Control Requirements for Future Air Force Systems", in Multi-Acess Computing: Modern Research and Requirements, Rochelle Park, NJ; Hayden, 1974, pp 17-29.

3. Barry W. Boehm, Robert K. McClean, and D. B. Urfrig, "Some Experiences with Automated Aids to the Design of Large-Scale Reliable Software", IEEE Trans. on Software Engineering, Vol. SE-1, No. 1, March 1975, pp. 125-133.

4. Barry W. Boehm, "Software Engineering", IEEE Trans. on Comp., Vol. C25, No. 12, Dec. 1976, p. 1226.

5. T. E. Bell, D. D. Bixler and M. E. Dyer, "An Extendable Approach to Computer-Aided Software Requirements Engineering", IEEE Trans. on Software Engineering, Vol. SE-3, No. 1, Jan. 1977.

6. F. T. Baker, "Chief Programmer Team Management of Production Programming", IBM Syst. J., No. 1, 1972.

7. E. W. Dijkstra, "The Humble Programmer", 1972 Turing Award Lecture, Comm. of ACM, October 1972

8. John D. Gannon and J. J. Horning, "Language Design for Programming Reliability", IEEE Trans. on Software Engineering, Vol. SE-1, No. 2, June 1975

9. Donald I. Good, Ralph L. London, and W. W. Bledsoe, "An Interactive Program Verification System", IEEE Trans. on Software Engineering, Vol. SE-1, No. 1, March 1975, pp. 59-67.

10. M. Hamilton and S. Zeldin, "Higher Order Software - a Methodology for Defining Software", IEEE Trans. on Software Engineering, Vol. SE-2, No. 1, March 1976, pp. 9-31.

11. C. A. R. Hoare, "The Engineering of Software: A Startling Contradiction", Computers and People.

12. M. A. Jackson, Principles of Program Design, Academic Press, N. Y., 1975.

13. H. F. Ledgard, "The Case for Structured Programming", Bit. Vol. 13, 1973, pp. 45-47.

14. C. R. Litecky and G. B. Davis, "A Study of Errors, Error-Proneness, and Error Diagnosis in COBOL", Comm. of ACM, Vol. 19, No. 1, Jan. 1976.

15. Clement McGowan, "Structured Programming: A Review of Some Practical Concepts", Computer, June 1975, pp. 25-30.

16. Lawrence J. Peters and Leonard L. Tripp, "Comparing Software Design Methodologies", Datamation, Vol. 23, No. 11, November 1977, pp. 89-94.

17. A. S. Philippakis, "Programming Language Usage", Datamation, Vol. 19, No. 10, Oct. 1973, pp. 109-114.

18. Noah Prywes, "Preparing for Future Needs", Computer, June 1975, pp. 70-72.

19. C. V. Ramamoorthy and S. F. Ho, "Testing Large Software with Automated Software Evaluation Systems", IEEE Trans. on Software Engineering, Vol. SE-1, No. 1, March 1975, pp. 46-58.

20. W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured Design", IBM Syst. J., No. 2, 1974.

21. Weinberg, et. al. "MetaCOBOL", High-Level COBOL Programming, Winthrop, Pa., c1977.

22. Rensis Likert, The Human Organization; Its Management and Value, McGraw Hill, 1967.

APPENDIX A

Product-Moment Method of Computing the Coefficient of Correlation

The coefficients of correlation computed in analyzing the data for this report were found by the Product-Moment method. These are the steps that were followed:

1. A scattergram is drawn for the two variables being correlated (No. of development problems and no. of structured techniques used, for example). A correlation table is drawn from the scattergram by writing the total occurences in each cell and then adding the rows and columns. The sums of the columns are called fx and of the rows, fy.

2. A mean for the rows and for the columns is assumed. For example, the mean number of problems was taken to be 1.97 and the mean number of techniques 2.24. Then x' is taken to be the number of columns away from the column in which the mean number of problems resides, and y' the number of rows from the mean techniques' row. The prime (') indicates distance from assumed means.

3. The fx', fy', fx'^2, and fy'^2 rows and columns are filled in and totaled. Therefore, $fx' = f \cdot x'$, $fy' = f \cdot y'$, $fx'^2 = fx' \cdot x'$, and $fy'^2 = fy' \cdot y'$. The fx' and fy' will give the correction to the assumed means in units of interval, and the standard deviations (σx and σy) may be computed from fx'^2 and fy'^2 and their corresponding corrections. The formulas are:

$$(1) \qquad cx = \frac{\Sigma fx'}{N}$$

where cx = correction to assumed mean

$\Sigma fx'$ = sum of (column deviations x occurences)

N = sample size

$$(2) \qquad \sigma = i \sqrt{\frac{\Sigma f x'^2}{N} - c^2}$$

where σ = standard deviation

i = interval size

f = frequency

x' = deviation in intervals

fx' = $f \cdot x'$

fx'^2 = $fx' \cdot x'$

$\Sigma x'^2$ = sum of (frequencies x squares of deviations)

N = sample size

c^2 = square of correction to assumed means

4. The sum of the product deviations, $\Sigma x'y'$ is computed. First, $\Sigma x'$ is found by multiplying the occurences in each cell by the number of columns it is from the assumed mean. When these are summed, the total ($\Sigma x'$) should equal fy'. $\Sigma y'$ is found in an analogous manner, multiplying the occurences in each cell by the no. of rows from the assumed mean. Upon summation, $\Sigma y'$ should equal fx'. $\Sigma x'y'$ may then be computed twice, first by the summation of the products $\Sigma x' \cdot y'$ and then the summation of the products $\Sigma y' \cdot x'$. The same result should be found by both methods.

5. The coefficient of correlation, r, is then computed:

$$r = \frac{\frac{\Sigma x'y'}{N} - cxcy}{\sigma' x \, \sigma' y}$$

where r = coefficient of correlation

$\Sigma x'y'$ = sum of product deviations

N = sample size

cx, cy = corrections to assumed means

$\sigma' x, \sigma' y$ = standard deviations in units of class interval

The level of significance was drawn from a table prepared by Garrett and Woodworth in Statistics in Psychology and Education, David McKay Co., Inc., N. Y., Fifth edition, June 1964, page 201. A small portion is reproduced:

Degrees of freedom (N - 2)	Level of Significance .05	.01
30	.349	.449
35	.325	.418
40	.304	.393

In this report the sample size was 37, so from the above, it may be seen that r must be .325 or greater to be at the .05 level of significance and .418 or greater to be at the .01 level.

APPENDIX B

Coefficient of Correlation between No. of Problems and No. of Structured Techniques Used.

No. of Problems (m= 2.0)

No. of Struct. Tech's Used (m= 2.2)	0	1	2	3	4	5	6	fy	y'	fy'	fy'^2	Σ x'	Σ x'y'
7				1				1	5	5	25	1	5
6								0	4	0	0	0	0
5		1	2					3	3	9	27	-1	-3
4	1		3	2	1			7	2	14	28	2	4
3			2		1			3	1	3	3	2	2
2	2	5	1	1			1	10	0	0	0	-4	0
1	1	2						3	-1	-3	3	-4	4
0	3	3	2	2				10	-2	-20	40	-7	14
fx	7	11	10	6	2	0	1	37		8	126	-11	26
x'	-2	-1	0	1	2	3	4						
fx'	-14	-11	0	6	4	0	4	= -11					
fx'^2	28	11	0	6	8	0	16	= 69					
Σ y'	-5	-5	10	5	3	0	0	= 8					
Σx'y'	10	5	0	5	6	0	0	= 26					

$cx = -.30$ $\quad cy = .22$

$cx^2 = .0900$ $\quad cy^2 = .0467$

$\sigma x = 1.83$ $\quad \sigma y = 1.33$

$r = .316$

APPENDIX C

Coefficient of Correlation Between Problems Ranking and Structured Techniques Ranking.

Tech'ls Used Ranking (m= 5.3)

Problems Ranking (m= 7.2)

	0-2	3-5	6-8	9-11	12-14	15-17	18-20	21-23	fy	y'	fy'	fy'^{2}	Σx'	Σx'y'
12-13				1					1	4	4	16	1	4
10-11	1		3	4			1		9	3	27	81	6	18
8-9			1	1					2	2	4	8	1	2
6-7			1	1				1	3	1	3	3	6	6
4-5	3	1	4						8	0	0	0	-7	0
2-3	1	1	1		1				4	-1	-4	4	-1	1
0-1	1	2	6		1				10	-2	-20	40	-2	4
fx	6	4	16	7	2	0	1	1	37		14	152	4	35
x'	-2	-1	0	1	2	3	4	5						
fx'	-12	-4	0	7	4	0	4	5	= 4					
fx'^{2}	24	4	0	7	8	0	16	25	= 84					
Σ y'	0	-5	-1	19	-3	0	3	1	= 14					
Σ x'y'	0	5	0	19	-6	0	12	5	= 35					

$cx = .11$ $cy = .38$

$cx^2 = .0121$ $cy^2 = .1444$

$\sigma x = 4.5$ $\sigma y = 3.98$

$r = .303$

Problem Rankings used:	
Cost/Time Overruns	- 6
Poor Communications	- 4
Personnel Turnover	- 5
Unreliable Hardware	- 1
Inadequate Devel. Tools	- 2
Lack of Ed. or Exp.	- 3
Other	- 1

Technique Rankings:	
Top-down Design	- 4
Step-wise Refinement	- 4
Hierarch. Ctl. Struct.	- 3
Modular Programs	- 2
Use of Comments	- 1
Other	- 1

APPENDIX D

Coefficient of Correlation Between Problems Ranking and No. of Language Improvements Desired.

Problems Ranking (m= 7.2)

No. of Lang. Imp's Desired (m=1.2)	0-4	5-9	10-14	15-19	20-24	fy	y'	fy'	fy'^{2}	Σ x'	Σ x'y'
5		1			1	2	4	8	32	3	12
4			1	1		2	3	6	18	3	9
3		2	1			3	2	6	12	1	2
2	1	2	1			4	1	4	4	0	0
1	2	7	2			11	0	0	0	0	0
0	6	8	1			15	-1	-15	15	-5	5
fx	9	20	6	1	1	37		9	81	2	28
x'	-1	0	1	2	3						
fx'	-9	0	6	2	3 = 2						
fx'^{2}	9	0	6	4	9 = 28						
Σ y'	-5	2	5	4	3 = 9						
Σ x'y'	5	0	5	6	12 = 28						

$c_x = .05$ $\qquad$ $c_y = .24$

$c_x^2 = .0025$ $\qquad$ $c_y^2 = .0576$

$\sigma x = .87$ $\qquad$ $\sigma y = 7.30$

$r = .600$

Problems Ranking used:

See Appendix C

APPENDIX E

(Case A)

Coefficient of Correlation Between No. of Structured Programming Techniques Used and No. of Language Improvements Desired (CASE, in-line loop, and Local Data).

No. of Lang. Imp's Desired (m= .5)	No. of Sturctured Prog. Tech's Used (m= 1.5) 0	1	2	3	fy	y'	fy'	fy'^{2}	Σx'	Σ x'y'
3		1		1	2	3	6	18	2	6
2	1		1	2	4	2	8	16	4	8
1	1	1		1	3	1	3	3	1	1
0	9	4	8	7	28	0	0	0	13	0
fx	11	6	9	11	37		17	37	20	15
x'	-1	0	1	2						
fx'	-11	0	9	22	= 20					
fx'^{2}	11	0	9	44	= 64					
Σ y'	3	4	2	8	= 17					
Σ x'y'	-3	0	2	16	= 15					

$cx = .54$ $\qquad cy = .46$

$cx^2 = .2922$ $\qquad cy^2 = .2111$

$\sigma x = 1.20$ $\qquad \sigma y = .89$

$r = .322$

(Case B)

Coefficient of Correlation Between No. of Structured Programming Techniques Used and All Language Improvements.

No. of Structured Prog. Tech's Used (m= 1.5)

No. of ALL Lang. Imp's Des. (m= 1.2)

	0	1	2	3	fy	y'	fy'	fy'^2	Σx'	Σx'y'
5		1		1	2	4	8	32	2	8
4				2	2	3	6	18	4	12
3	1	1		1	3	2	6	12	1	2
2	3		1	1	5	1	5	5	0	0
1	4	1	1	3	9	0	0	0	3	0
0	3	3	7	3	16	-1	-16	16	10	-10
fx	11	6	9	11	37		9	83	20	12
x'	-1	0	1	2						
fx'	-11	0	9	22	= 20					
fx'^2	11	0	9	44	= 64					
Σy'	2	3	-6	10	= 9					
Σx'y'	-2	0	-6	20	= 12					

$cx = .54$ $cy = .24$

$cx^2 = .2916$ $cy^2 = .0576$

$\sigma x = 1.20$ $\sigma y = 1.48$

$r = .110$

APPENDIX F

Other Coefficient of Correlation Calculations

Chapter 3 makes reference to certain correlation investigations of shop sizes and language improvements, structured techniques used, and software development problems. The correlation tables and coefficients of correlation that were computed are included below; detailed calculations are omitted but should be readily obtainable by the method outlined in Appendix A.

Table F-I - Correlation between the Use of COBOL and the Desire for Language Improvements

Language Used (m= COBOL)	Language Improvements Desired? (m= yes)		
	no	yes	
COBOL	10	21	
Other	5	1	r = .384

Table F-II - Correlation between Shop Size and the Desire for More Accurate Error Diagnostics

Shop Size (m= MD)	Better Diagnostics Desired? (m= no)		
	no	yes	
LG	4	3	
MD	9	6	
SM	11	4	r = .141

Table F-III - Correlation between Shop Size and Desire for CASE Statement

	CASE Statement Desired? (m= no)		
	no	yes	
LG	5	2	
MD	13	2	
SM	12	3	r = .051

Table F-IV - Correlation between Shop Size and Desire for In-Line Looping (DO UNTIL or WHILE)

	In -Line Looping Desired? (m= no)		
	no	yes	
LG	7	0	
MD	11	4	
SM	13	2	r = -.069

Table F-V - Correlation between Shop Size and Desire for Cross-reference of CALL'ed Procedures

	CALL cross-reference Desired? (m= no)		
	no	yes	
LG	5	2	
MD	9	6	
SM	13	2	r = .182

Table F-VI - Correlation between Shop Size and the Desire for Local Data

	Local Data Desired? (m= no)		
	no	yes	
LG	6	1	
MD	14	1	
SM	13	2	r = -.012

Table F-VII - Correlation between Shop Size and the Use of Top-Down Design

	Top-Down Design Used? (m= no)		
	no	yes	
LG	3	4	
MD	8	7	
SM	12	3	r = .311

Table F-VIII - Correlation between Shop Size and the Use of Comments

	Comments Used? (m= no)		
	no	yes	
LG	3	4	
MD	9	6	
SM	11	4	r = .235

Table F-IX - Correlation between Shop Size and Cost/Time Overruns

	Cost/Time Overruns? (m= no)		
	no	yes	
LG	2	5	
MD	10	5	
SM	12	3	r = .369

Table F-X - Correlation between Shop Size and Lack of Education or Experience

	Lack of Education or Experience? (m= no)		
	no	yes	
LG	6	1	
MD	6	9	
SM	8	7	r = -.165

APPENDIX G

Data From Survey Responses

	IND	SH SZ	P S	M S	S S	SH TP	PRIM. LANG.	NEW LANG.	LANG. IMP.	STRCT. TECHS.	SW. DEV. PROBLS.
1	AIR	LG	M	Y	Y	B	ALC	NO	NO	NO	NO RESP
2	COM	LG	M	Y	Y	B	COBOL	COBOL	ACE	ACDE;W	AB
3	ED	SM	M	Y	Y	BN	COBOL FORTRN	NO	AD	CD	C
4	ED	SM	M	Y	Y	B	COBOL	COBOL	NO RS	C	NO RESP
5	ED	SM	M	Y	Y	B	COBOL	NO	NO	CE	NONE
6	ED	SM	M	Y	Y	B	COBOL	NO	E	C	F
7	ENG	SM	M	Y	Y	B	COBOL	NO	AB	NO	F
8	ENG	MD	M	N	Y	BN	COBOL FORTRN	U/F	B C/F	NO	A
9	ENT	SM	M	Y	Y	B	COBOL	IMPRS	O	NO!!	F
10	GRN*	SM	M	Y	Y	B	COBOL	COBOL	ABCDE	AD	ABCDEF
11	GRN*	SM	M	Y	Y	B	RPG II	NO	NO	ACDE;DD	CF
12	HOS	SM	M	Y	Y	B	COBOL	ANS COBOL	CE	NO	NO RESP
13	HOS	SM	M	Y	Y	B	NEAT/3	NO	NO	CD	E
14	HOS	MD	M	Y	Y	B	COBOL MARK IV	NO	C;RG	ACE	CD
15	INS	MD	M	Y	Y	B	COBOL	COBOL	NO	YES	PD
16	INS	MD	M	Y	Y	B	COBOL	NO	ABCE	ACDE	ABCF
17	INS	MD	M	Y	Y	B	COBOL	NO	NO	C	SS
18	INV	MD	S	Y	Y	B	COBOL	RPG II	O	NO	CF
19	MFG	MD	M	Y	Y	B	COBOL	COBOL	NO RS	ADE	CF
20	MFG	LG	M	Y	Y	B	COBOL	NO	E	ACDE	ADF
21	MFG	SM	M	Y	Y	B	COBOL	NO	NO	YES	NO RESP
22	MFG	MD	M	Y	Y	B	COBOL	NO	E;O	NO	CF
23	MFG	LG	M	Y	Y	B	COBOL	NO	NO	YES	A
24	MFG	MD	M	Y	Y	B	MARK IV RPG II	COBOL	NO	NO	BE;PD

25	MFG	SM	M	Y	Y	B	COBOL	NO	NO	NO	NONE
26	MFG	LG	M	Y	Y	B	COBOL	UFO/FOCUS	NO	ACDE	U/S CODE
27	MFG	LG	M	Y	Y	B	ANS-COBOL	ANS COBOL	ACDE	ACDE	AB
28	MFG	LG	M	Y	Y	B	COBOL	NO	RG;SN	NO	ABE
29	R/D	MD	M	Y	Y	B	COBOL ALC	COBOL	BCE	A;W	BF
30	R/D	SM	M	Y	Y	B	COBOL	NO	NO	ACDE	CF
31	SWD	MD	M	N	Y	B	MIIS	COBOL	NO	AD;WT;W	AF
32	FED	MD	M	Y	Y	B	COBOL	NO	CE	ACDE	ABD
33	UTL	MD	M	Y	Y	B	COBOL	NO	BCE	CE	BF
34	UTL	MD	M	Y	Y	B	COBOL	NO	NO	YES	ABF
35	UTL	SM	S	Y	Y	B	COBOL	NO	E	CDE	AF;SS;PD
36	UTL	MD	M	Y	Y	B	COBOL	MAPPER	ABCDE	ABCDE; WT;HIPO	CF;NS
37	VDR*	SM	M	Y	Y	B	RPG II	RPG III	RF	ABCDE	A

* In body of report, was included with retail/distribution.

Summary of Abbreviations

1. IND = Industry;
   - AIR = Airline
   - COM = Communication
   - ED = Education
   - ENG = Engineering Consultant
   - ENT = Entertainment
   - GRN = Grain
   - HOS = Health Care
   - INS = Insurance
   - MFG = Manufacturing
   - R/D = Retail/Distribution
   - SWD = Software Development
   - FED = Federal Government (Dept. of Agriculture)
   - UTL = Utility
   - VDR = Veterinary Drug Research/Mfg/Distr.

2. SH SZ = Shop Size;
   - LG = Large
   - MD = Medium
   - SM = Small

3. PS = Position of Respondent in Department
   - M = Management
   - S = Staff

4. MS = Management Satisfied with Language?;
   - Y = Yes (all except 2 were yes)
   - N = No

5. SS = Staff Satisfied with Language?;
   - Y = Yes (all were yes)
   - N = No

6. SH TP = Shop Type;

   B = Business (all except 2 were Business)

   N = Numeric

   BN = Both (only 2 were both)

7. PRIM. LANG. = Primary Language;

8. NEW LANG. = New Language used for new applications or for converting old ones;

   U/F = User Friendly (probably doesn't exist)

9. LANG. IMP. = Language Improvements Desired;

   A = CASE statement

   B = In-line looping (DO UNTIL or WHILE DO)

   C = 'CALL' cross reference

   D = Local Data in procedures

   E = More accurate error diagnostics

   B C/F = Blend of COBOL and FORTRAN

   NO RS = No response

   RF = More flexibility in reading files

   RG = Easier report generation

   SN = Simplified nesting

   O = Other

10. STRCT. TECHS. = Structured Techniques used;

   A = Top-down design

   B = Step-wise refinement

   C = Hierarchical Control Structures (perform, gosub, etc.)

   D = Modualr programs/segments

   E = Comments

   DD = Data Dictionary

   HIPO = HIPO Diagrams

   W = Warnier-Orr Diagrams

   WT = Structured Walk-thru

11. SW. DEV. PROBLS. = Software Development Problems

A	=	Cost/time overruns
B	=	Poor communications with users
C	=	Personnel turnover
D	=	Unreliable hardware
E	=	Inadequate development tools (text editors, libraries, etc.)
F	=	Lack of education or experience
NS	=	No standards
PD	=	Poor documentation
SS	=	Small staff
U/S CODE	=	Old unstructured code

SOFTWARE DEVELOPMENT: A SURVEY OF CURRENT PRACTICES

by

LOYE E. HENRIKSON

B. Sc. Ed., Central Missouri State University, 1974

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1982

In recent years, rising software costs have prompted the development of many techniques and automated aids for the software development cycle. A few examples would include software engineering, structured design and programming, language preprocessors, and automated software testing and evaluation. To determine the extent of usage and effectiveness of new techniques and automated aids, a questionnaire was sent to 58 data processing shops in the Kansas City area.

COBOL was found to be the most used language; 84% of respondents used it. A high degree of satisfaction with programming languages was expressed, although a few improvements were desired: primarily compiler improvements of error diagnostics and a cross reference of called procedures. A consistent and fairly high usage of structured techniques was exhibited by small, medium, and large shops. Software development problems were also extensively reported, in spite of the use of structured techniques.

A small positive correlation (although not a statistically significant one) was found between the use of structured techniques and the occurence of software development problems. A strong positive correlation was found between the occurence of problems and the desire for language improvements.

It was concluded that another, more quantitatively designed questionnaire should be sent to a wider sample group before any firm judgements were made on the effectiveness of newly developed techniques.