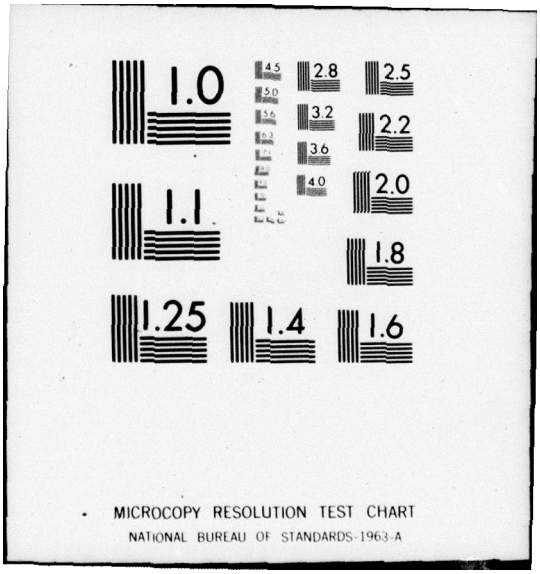
AD-A041 645 IBM FEDERAL SYSTEMS DIV GAITHERSBURG MD F/G 9/2
AN APPLICATION OF FORMAL INSPECTIONS TO TOP-DOWN STRUCTURED PRO--ETC(U)
JUN 77 M P PERRIENS F30602-76-C-0166
UNCLASSIFIED RADC-TR-77-212 NL

1 OF 1
AD A041645

END
DATE FILMED
8 - 77


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1
8.0
9.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AD A041645

12
B.S.

RADC-TR-77-212
Final Technical Report
June 1977

# AN APPLICATION OF FORMAL INSPECTIONS TO TOP-DOWN STRUCTURED PROGRAM DEVELOPMENT

IBM Corporation

Approved for public release; distribution unlimited.

DDC
RECEIVED
JUL 18 1977
C

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

AD No.
DDC FILE COPY

This report has been reviewed by the RADC Information Office (OI) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public including foreign nations.

This report has been reviewed and is approved for publication.

APPROVED: Joseph L Femia

JOSEPH FEMIA
Project Engineer

APPROVED: Robert D. Krutz

ROBERT D. KRUTZ, Colonel, USAF
Chief, Information Sciences Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

Do not return this copy. Retain or destroy.

# *MISSION*
# *of*
# *Rome Air Development Center*

*RADC plans and conducts research, exploratory and advanced development programs in command, control, and communications ($C^3$) activities, and in the $C^3$ areas of information sciences and intelligence. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.*

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

| REPORT DOCUMENTATION PAGE | READ INSTRUCTIONS BEFORE COMPLETING FORM |
|---|---|
| 1. REPORT NUMBER: RADC-TR-77-212 | 2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER |
| 4. TITLE (and Subtitle): AN APPLICATION OF FORMAL INSPECTIONS TO TOP-DOWN STRUCTURED PROGRAM DEVELOPMENT | 5. TYPE OF REPORT & PERIOD COVERED: Final Technical Report |
| | 6. PERFORMING ORG. REPORT NUMBER: N/A |
| 7. AUTHOR(s): Matthew P. Perriens | 8. CONTRACT OR GRANT NUMBER(s): F30602-76-C-0166 |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS: IBM Corporation, Federal Systems Division, 18100 Frederick Pike, Gaithersburg MD 20760 | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 63728F, 55580836 |
| 11. CONTROLLING OFFICE NAME AND ADDRESS: Rome Air Development Center (ISIM), Griffiss AFB NY 13441 | 12. REPORT DATE: June 1977 |
| | 13. NUMBER OF PAGES: 74 |
| 14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Same | 15. SECURITY CLASS. (of this report): UNCLASSIFIED |
| | 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A |

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

DDC RELEASED JUL 18 1977

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Joseph F. Femia (ISIM)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Top-Down Structured Programming
Formal Inspections

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

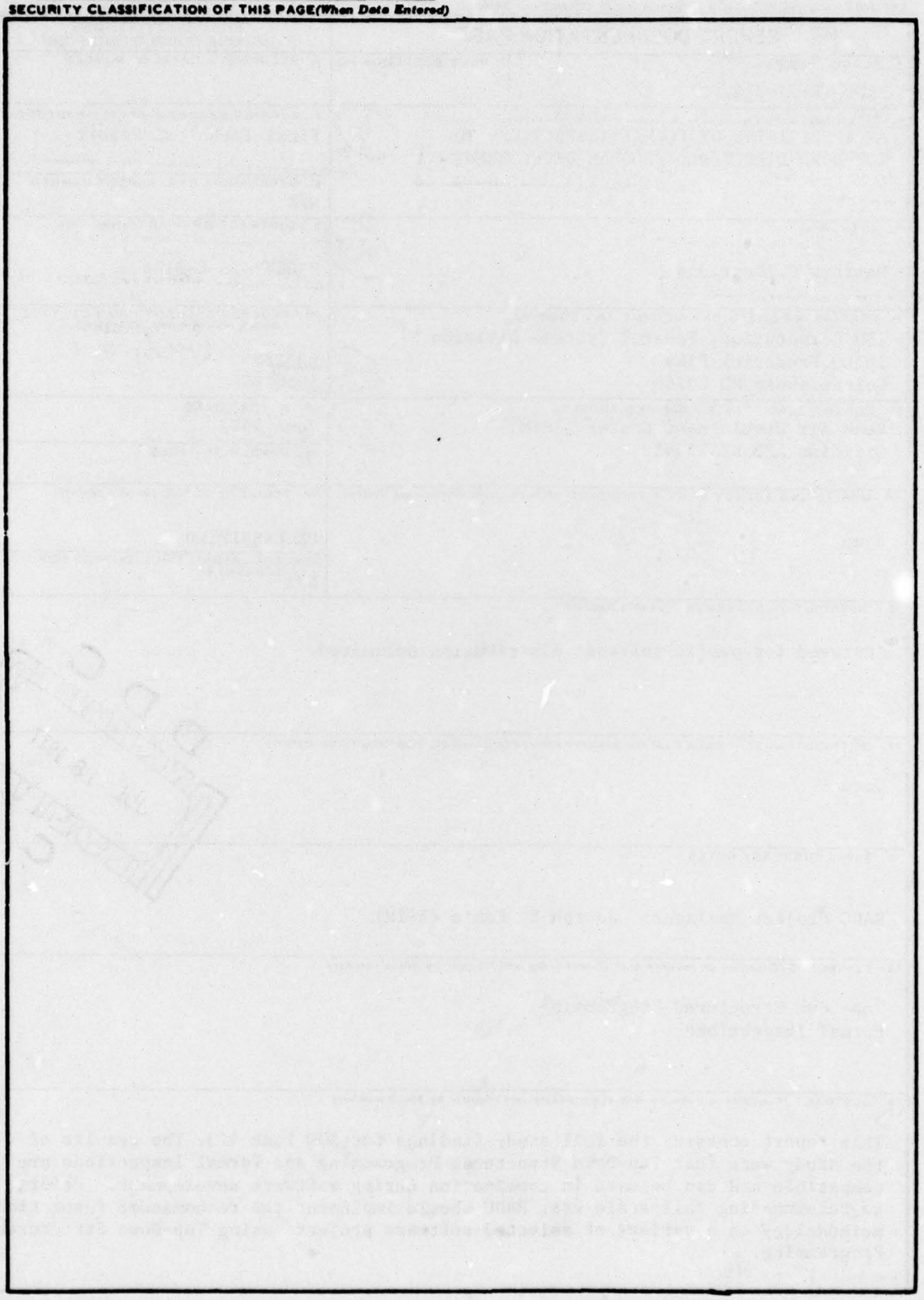
This report contains the full study findings for SOW Task 4.3. The results of the study were that Top-Down Structured Programming and Formal Inspections are compatible and can be used in combination during software development. Prior to recommending full-scale use, RADC should implement the recommended inspection methodology on a variety of selected software projects using Top-Down Structured Programming.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

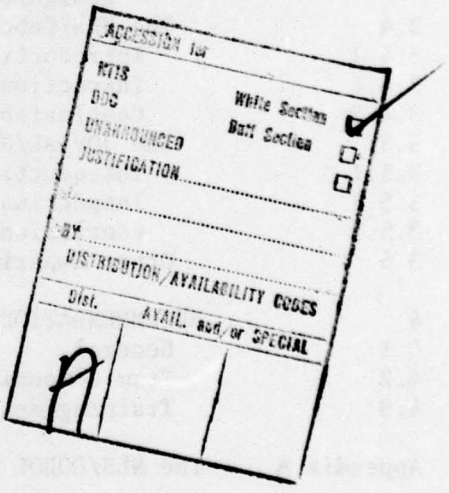
174 950



## PREFACE

This report was produced by IBM in response to Task 4.3 in the Statement of Work for Design/Code Verification under contract number F30602-76-C-0166. It is delivered to RADC in accordance with item A003 of the Contract Data Requirements List.

The report was prepared by Matt Perriens, with significant contributions by Dick Kopp. Consultation and review were provided by Joe Femia, Project Engineer, and John McNamara at RADC; Donna Campbell, Mike Fagan, Andy Ferrentino, Marvin Kessler, Harlan Mills, and Earl Stroup at IBM; and Orville Goering at Intech.



CONTENTS

## ILLUSTRATIONS



## TABLES

## EVALUATION

The principle objective of this effort is the integration of design/code inspections in a Top Down Structured Programming (TDSP) environment.

The basic objective of inspection is to detect and remove errors and ambiguities as early as possible in the software development process. This results not only in improved software quantity, but also in project development cost savings.

Inspection methodologies developed were utilized on two experimental software projects. The inspection of design and code were carried out at definite points in the software development process, utilizing precise procedures, various checklists, and exit criteria.

As a results of this effort, it has been determined that formal inspections are compatible with TDSP and can be accomplished effectively by utilizing teleprocessing facilities when face to face inspections are not possible because of involvement of experts located at different distant locations.

Guidelines were established for inspection team composition, training requirements of team members, and checklists to be utilized in the inspection process.

Joseph F Femia

JOSEPH FEMIA
Project Engineer

# Section 1. INTRODUCTION

## 1.1 BACKGROUND

In recent years considerable emphasis has been placed upon improving the effectiveness of the processes involved in the development, maintenance and documentation of computer software. IBM has participated in the formulation, use, and assessment of new and refined methodologies.

The 15-volume Structured Programming Series[1] was developed for RADC by IBM in 1974. Volume 15, Validation and Verification Study, reported on the techniques currently used for verifying computer programs and software systems, and the effect that structured programming technology has on these techniques.

The extensive use of formal design and code inspections[2], developed and tested by Michael Fagan and others[2], have also been reported upon. However, the formal inspection techniques were not developed using top-down structured programming (TDSP) and no reports have been published which describe the effects of combining these methodologies in a software development project. These methodologies are both directed towards reducing the cost of implementing software and improving the reliability of the software product.

## 1.2 PROJECT PURPOSE

Since the concept of formal inspections was developed largely in parallel with the implementation of TDSP it became a matter of interest to determine whether these methodologies could be used in combination, and whether formal inspections ought to be made a part of RADC's software development and procurement practices.

IBM's assignment for this project was to examine the objectives, assumptions, and methods of TDSP and informal inspections, to develop an integrated software development methodology based upon our experiences with two experimental software development projects, and to make recommendations to RADC regarding the combined use of these methodologies.

Both TDSP and formal inspections seek to facilitate the production of programs that are correct. TDSP is a program design and implementation methodology which accomplishes correctness through recursive verification of incremental

---

[1] Copies of individual volumes or the complete set may be obtained from the National Technical Information Service (NTIS), 5285 Port Royal Road, Springfield, Virginia 22161, by referencing RADC-TR-74-300.

[2] Reference several IBM technical reports and an IBM Systems Journal article as shown in the Bibliography.

levels of design and code. Formal inspection is a method of controlled inspection of design and code, carried out at defined points in the software development process, with clear and precise procedures, checklists, and exit criteria.

There is less than general agreement on what phases or tasks the software development process encompasses. Formal inspection assumes at least three phases:

- o High-level design
- o Low-level design
- o Coding, testing, and integrating.

TDSP focuses primarily on the latter two of these three.

## 1.3 PROJECT DESCRIPTION

The principal objective of the study leading to this report was to develop a workable integration of formal inspections and TDSP. To accomplish this, we studied current verification techniques, formulated a preliminary plan for integrating formal inspections and TDSP, and implemented that plan (with inevitable modifications) on the development of two experimental software products: a COBOL precompiler and a JOVIAL/JOCIT precompiler.

The conclusions and recommendations presented here are based mostly on a comparative analysis of the methodologies themselves. For reasons explained in greater detail throughout this report, the data gathered from the two experimental tasks were insufficient to provide statistical support for the conclusions. The experimental conclusions and recommendations that do appear here should, therefore, be viewed as tentative.

## 1.4 CONCLUSIONS

a. Formal inspections are basically compatible with TDSP. They have common objectives, but their methodologies show occasional differences in scope, timing, and focus. To integrate these methodologies, appropriate modifications will have to be made to each.

b. High-level design is not addressed specifically by TDSP. Hence, there is no conflict between the methodologies at this level. Formal inspections could benefit, however, from better high-level design methodologies. This is an area for further exploration.

c. Successful application of formal inspections in a TDSP environment depends critically on total management commitment, close adherence to formal inspection procedures[1], and the precise formulation and application of exit criteria for tasks being inspected.

d. TDSP and formal inspections are complementary in the sense that TDSP includes the development of design, code, and documentation which are highly structured and therefore, easier to read, comprehend, and verify. Training in the verification oriented disciplines of TDSP is likely to enhance the effectiveness of all who participate in the formal inspections.

e. Formal inspections can be made more effective through the use of good supporting documentation and visual aids. Such aids should clearly present the principal model elements, their system representations, as well as their interrelations.

f. It is important for moderators to undergo specialized training in planning and conducting formal inspections. Such training, as provided by IBM, is directed toward the mechanics as well as the human relations aspect of reviews and inspections.

g. By their nature, formal inspections are likely to be most effective on large projects. The validity and pertinence of data collected and fed back into other subtasks of the developing system derive largely from size -- from the weight of numbers and the likelihood that an analyzed activity will recur.

h. Teleprocessing facilities may provide a welcome alternative to face-to-face inspections on projects which require short-time involvement of experts from distant locations. Our own experiment, using the NLS (online system) facilities for the COBOL precompiler code inspection, yielded very good results (see Appendix A).

## 1.5 SUMMARY RECOMMENDATIONS

a. In order to obtain statistical evidence of the cost-effectiveness of formal inspections, RADC should implement the recommended inspection methodology on a variety of operational software projects using TDSP.

b. High-level inspections ($I_0$) should be conducted according to existing formal inspection guidelines. Design ($I_1$) and code ($I_2$) inspections can be conducted separately or in combination depending on prevailing conditions.

---

[1] For a survey of the technique, samples of records and checklists, as well as further references, see Fagan, M. E., "Design and Code Inspections to Reduce Errors in Program Development," IBM Systems Journal, Vol. 15, No. 3, pp. 182-211.

c. Except for certain types of problems, high-level design (transforming a problem description into a program structure) is less well-defined and developed than program design and implementation. The following facets need to be addressed:

   1. Defining and delimiting the scope of all phases and subtasks comprising the software development process

   2. Establishing appropriate levels of explicitness, completeness, etc., for each of these subtasks, with emphasis on the high-level design phase.

   3. Deriving pertinent inspection criteria for each of these subtasks, to augment the set currently being used in formal inspections.

   4. Classifying existing design methodologies according to their scope, level of explicitness, and completeness.

d. Inspections should normally be carried out by a team of four members: the reviewee, two inspectors, and a moderator.

e. Training in formal inspections is required for persons who will serve as moderators. Training or experience in TDSP, emphasizing the verification and stepwise refinement methods, is required for everyone involved in the inspection process.

f. Vendors should submit their own inspection plan indicating

   - At which points inspections will be applied
   - What criteria will be applied
   - Inspection team makeup
   - Feedback path
   - Checklists.

   Proposals should clearly set forth the high-level design methodology vendors plan to use. In fact, the proposal itself should be developed according to this methodology.

## 1.6 REPORT ORGANIZATION

The first section of this report contains background information which led to this study, a description of the project and its purpose, conclusions, and a summary of recommendations.

Section 2 contains a concise review of TDSP concepts pertinent to this study.

Section 3 discusses the implications of implementing formal inspections in a TDSP environment, based upon our review of current literature and experience with the two experimental projects.

Section 4 expands upon the summary recommendations presented in the first section.

Appendices A and B contain the detailed description of experimental results obtained from the inspections performed upon the two precompiler tasks.

Appendix C contains working notes on the place and scope of design in the software development process and suggests areas for further exploration.

# Section 2. TOP-DOWN STRUCTURED PROGRAMMING

## 2.1 INTRODUCTION

The fundamentals of the structured programming technology were described in the 15-volume series on Structured Programming, produced by IBM, FSD, under U.S. Air Force (RADC) contract #F30602-74C-0168. (Reference Appendix B of Volume 15 of that series.) This description encompassed three related techniques:

a. Top-Down Structured Programming (TDSP)
b. Programming Support Libraries (PSLs)
c. Chief Programmer Team (CPT) operations.

Only the first of these techniques, TDSP, has a direct effect on the implementation of formal inspections. PSLs can facilitate such implementation by providing readable, structured documentation for use during inspections. The principal characteristics of TDSP are briefly reviewed below.

## 2.2 TDSP DEFINITION

TDSP is a methodology that includes the two interrelated techniques of structured programming (SP) and top-down programming (TDP).

SP is based on the well-known Structure Theorem of Boehm and Jacopini[1], which says that any proper program (a program having but one entry and one exit) is equivalent to a program containing only the three logic structures: SEQUENCE, IFTHENELSE, and WHILEDO (or DO WHILE).

These three structures are prime programs because they do not contain any subprograms that are proper programs. Prime programs can be viewed as the building blocks from which all compound programs are made. Conversely, they are the "primes" into which all proper compound programs can be "factored" (parsed).

These few concepts provide the basic tools for both the generation and analysis (parsing) of programs. The notion of a prime program is the key to the conceptualization of fundamental program design and provides the starting point for the process of stepwise refinement.

---

[1] Boehm, C. and Jacopini, J., "Flow Diagrams, Turing Machines and Languages With Only Two Formation Rules," CACM, Vol. 9, No. 5, May 1966, pp. 366-371.

Use of the basic logic structures is recommended because it yields a number of advantages, chief among which is a clear visibility of program structure, not only to the eye, but also to the mind. This, in turn, leads to greater manageability and, hence, verifiability of programs. Since verification constitutes our principal concern on this project, two related facts should be mentioned in connection with SP and formal inspections:

a. SP as a methodology incorporates a set of correctness proofs which enable the designer/programmer to confirm at each step of the program's development that no errors have been introduced.

b. Formal inspections begin with a high-level design ($I_0$) inspection, which is governed by external critera, i.e., it is primarily user oriented. The exit criteria applied in this inspection are such as to guarantee an adequate basis for the development of an internally (data processing) oriented low-level design and inspection ($I_1$).

The integration of the two methodologies is illustrated in Figure 1.

On the software development side, the high-level design activity is less well defined than low-level design, coding, and integration. This gap had to be filled to permit systematic development and inspection of the experimental programs called for in the contract.

On the inspection side detailed guidelines have been defined for all three kinds of inspection. They will be described in greater detail in Section 5 where we will address the problem of implementing these inspections in a top-down structured programming environment. At that point we will also reexamine the TDSP practices and suggest extensions or modifications needed to effect the integration of the two methodologies.

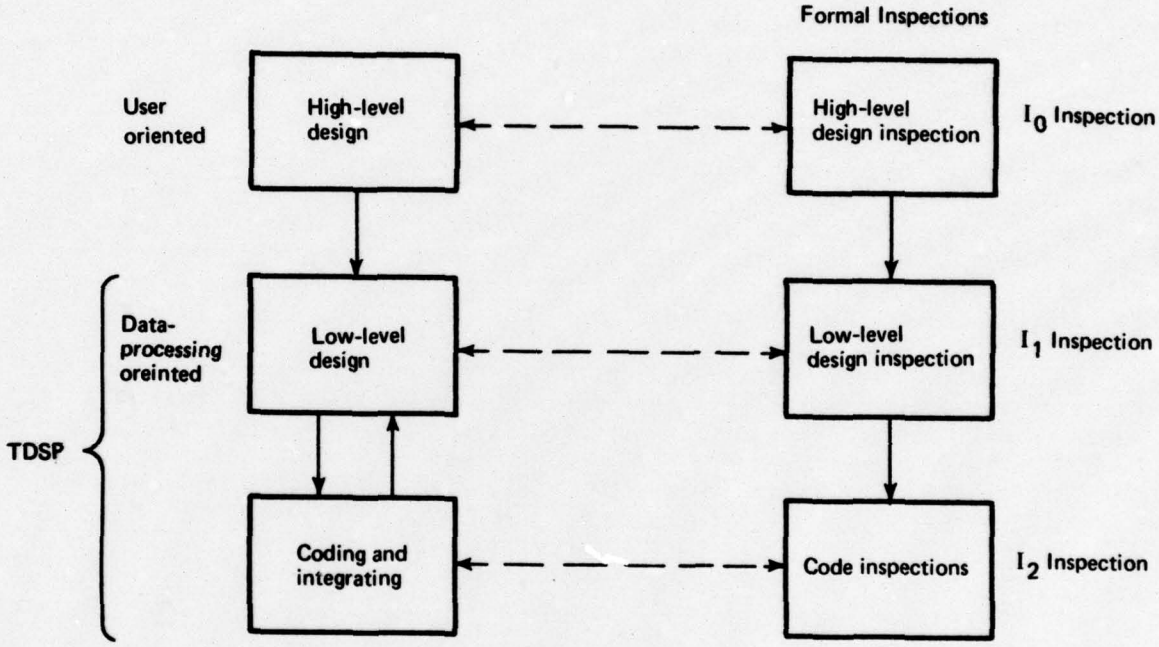



Figure 1. Software Development and Formal Inspections Under TDSP

# Section 3. IMPLEMENTING DESIGN/ CODE INSPECTIONS IN A TOP-DOWN ENVIRONMENT

## 3.1 INTRODUCTION

The principal objective of this project has been to investigate the feasibility of implementing formal inspections in a Top-Down Structured Programming (TDSP) environment.

Formal inspections aim at achieving improved quality and productivity through highly structured reviews of design and code. These inspections are held at predetermined stages of the software development process by a team consisting of a moderator, the developer, and one or more inspectors. Detailed error analysis and reporting procedures are a significant part of this review method.

When one speaks of reviews in a TDSP environment, one usually has in mind the so-called Structured Walk-Throughs. But such a comparison of formal inspections with Structured Walk-Throughs tends to obscure the fact that, in TDSP, the verification of programs is not separate from, but an integral part of, the very process that creates *those programs*. Consequently, a major part of the verification process is continual rather than periodic in TDSP.

Our own experience with formal inspections (as applied to design and code on two precompiler development tasks) indicates that it is, indeed, feasible to implement formal inspections in a TDSP environment. Some minor and some major problems must be faced.

These problems derive mainly from the following characteristics of the two methodologies under investigation:

a. In TSDP, designing and coding are alternating (recurring) rather than sequential (one-time) activities. As a consequence, design is not complete until the stepwise implementation process has reached the bottom level of the program structure. By then most of the coding has already been done.

b. Formal inspections address high-level design, low-level design, and code. The high-level design process, however, is not yet clearly defined in either scope or methodology. TDSP, in particular, lends most of its methodological support to program (low-level) design.

For all their differences, formal inspection and TDSP are also complementary in several ways. This makes their integration not only possible but also desirable.

## 3.2 FORMAL DESIGN/CODE INSPECTIONS

Formal inspections of design and code, as developed by Michael Fagan and his associates at IBM, can be characterized as follows:

a. Their primary purpose is to improve productivity by attaining ever-improving error rates.

b. These improvements are made possible by a systematic and efficient design and code verification process, with well-defined roles and procedures for inspection participants.

c. Another important factor in attaining these improvements is the manner in which inspection data is categorized and made suitable for process analysis.

d. The programming process is viewed as consisting of the following process operations and levels, with corresponding outputs and inspections:

| Process Operations | | Output (+ detailed exit criteria) | Inspections |
|---|---|---|---|
| DESIGN | Level 0 | Statement of Objectives | |
| | Level 1 | Architecture | |
| | Level 2 | External Specifications | |
| | Level 3 | Internal Specifications | ◀High-level design $I_0$ Inspection |
| | Level 4 | Logic Specifications | ◀Low-level design $I_1$ Inspection |
| CODE | Level 5 | Coding/Implementation Unit Test | ◀$I_2$ Code Inspection |
| TEST | Level 6 | Function Test | |
| | Level 7 | Component Test | |
| | Level 8 | System Test | |

Note: Exit criteria for a particular process operation are requirements that must be satisfied before the next process operation can be started. This involves a yes/no decision based on a review of major and minor errors recorded and the satisfactory completion of follow-up activities.

e. To allow inspectors to focus on but one objective at a time, formal inspections are carried out in a series of separate inspection operations:

| Inspection Operation | Purpose |
|---|---|
| 1. Overview | Communication/general education |
| 2. Preparation | Education (detailed) |
| 3. Inspection | Error finding |
| 4. Rework | Error fixing |
| 5. Follow-up | Error fixing control |

f. The developers of the formal inspection technique strongly emphasize the crucial importance of management commitment from the beginning of the project. This commitment includes prior scheduling and allocation of personnel and funds. Without this, inspections are all too likely to be deferred until disaster has struck. At that point, audits and inspections -- no matter how systematic -- amount to little more than autopsies.

Having an established process and formal procedure, inspections tend to vary less and produce more repeatable results than do walk-throughs. Although there appears to be a clear tendency toward more systematic and rigorous reviews -- even among those who would say they conduct walk-throughs -- Fagan and his associates were among the first to proclaim the benefits of systematic inspection-data gathering and analysis, and they supplied a methodology for attaining them.[1]

## 3.3 EXPERIMENTAL IMPLEMENTATION OF FORMAL INSPECTIONS AND TOP-DOWN DEVELOPMENT

### 3.3.1 Compatibility of the Two Methods

Before describing our application of formal inspections to our own development of two precompilers, we need to make a few preliminary observations pertinent to TDSP, formal inspections, and the software development process in general. (Some of the observations made here imply extensions of the SP methodology as described in the 15-volume SP series.)

a. It must be recognized that the term top-down development means different things to different people. (See, for example, "How Many Directions is Top-Down," by Dennis P. Galler, Datamation, June 1976.) This unfortunate fact makes effective communication difficult and casts doubt on the comparability of production and inspection data collected on projects that were reportedly done "top-down."

---

[1] See M. E. Fagan, "Design and Code Inspections to Reduce Errors in Program Development," IBM Systems Journal, Vol 15., No. 3, 1976, pp. 182-211.

b. This problem is complicated by the fact that there is less than total agreement on the sequence and nature of subtasks comprising the software life cycle or even the software development process. (See Appendix C.)

c. Low-level design inspections ($I_1$) and code inspections ($I_2$), involve the systematic review of products for which increasingly effective methodologies are being developed as part of the Modern Programming Practices repertoire. If by high-level design, however, we mean an externally (user) oriented activity that systematically turns a user's problem statement into a structure for the computer programs required to solve that problem, then we can find only scant technological support at the present time. Specifically, the top-down structured approach described in the SP series, covers only program design. This would be the appropriate subject matter for a low-level ($I_1$) inspection. The TDSP methodology assumes that some prior top-down decomposition process has produced a definition of all the modules of the program being developed, the relationships among modules (including interface specifications), and a description of the format of any common data sets. This is a high-level design activity, recorded in Process Design Language or in some other suitable language. Detailed design of individual modules is not envisaged at this stage. The structural decomposition of the program into its component modules can be shown hierarchically in the form of a decomposition tree. An example is shown in Figure 2.

   A representation in matrix form shows that several modules are called by more than one higher-level module (see Figure 3).

d. In TDSP the low-level design is not developed all at one time. Nor is the coding done all at one time. They are done in parallel. As a result, there is at every level a (recurring) cyclic activity of designing, coding, testing (and integrating). Coding of a module is not begun until the detailed design of its subsidiary modules has been completed. Indeed, since the code of a module may contain stubs for lower-level (called or included) code, one can say that the code is not fully defined without the design of the subsidiary modules.

e. Segments (and the modules built up from them) are implemented in the order of logic flow, so that the program, although incomplete at any of the intermediate stages of development, will always be executable (assuming appropriate stubs have been written). In practical situations, the implementation is frequently begun with a single selected path of the calling hierarchy -- either because that path is known to be critical, most complex, or simply because it implements a function that is easily recognizable by the intended user.

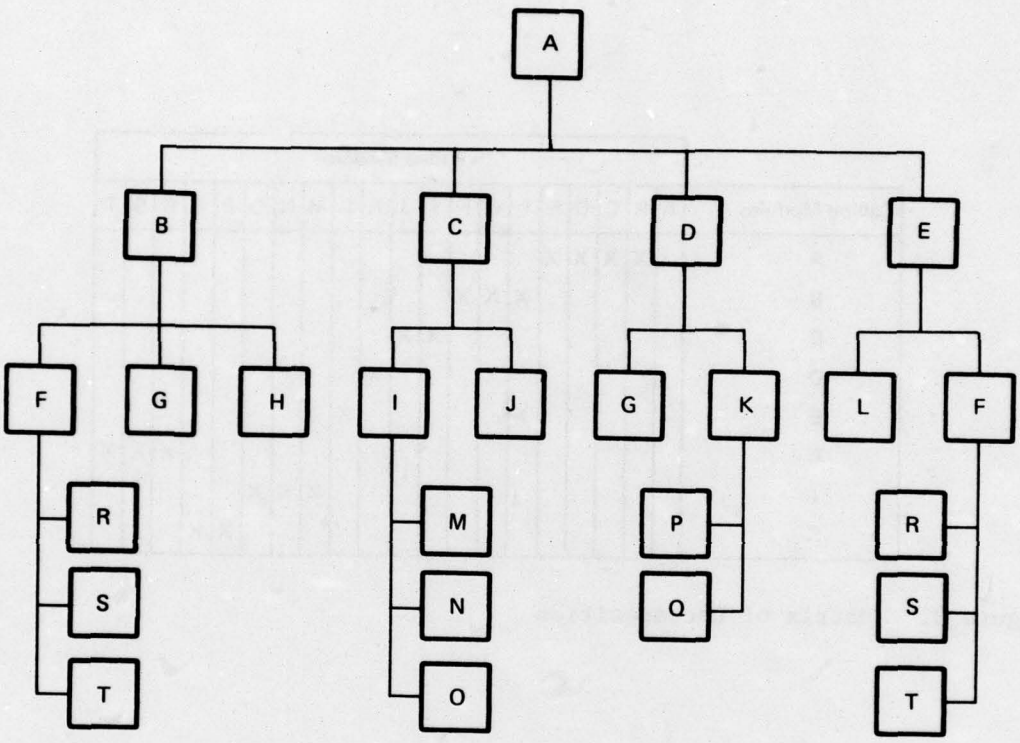



Figure 2. Decomposition Tree

| | Called Modules | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Calling Modules | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T |
| A | | X | X | X | X | | | | | | | | | | | | | | | |
| B | | | | | | X | X | X | | | | | | | | | | | | |
| C | | | | | | | | | X | X | | | | | | | | | | |
| D | | | | | | | X | | | | X | | | | | | | | | |
| E | | | | | | X | | | | | | X | | | | | | | | |
| F | | | | | | | | | | | | | | | | | | X | X | X |
| I | | | | | | | | | | | | | X | X | X | | | | | |
| K | | | | | | | | | | | | | | | | X | X | | | |

Figure 3. Matrix of Decomposition

f. The English-like PDL statements provide not only the high-level (and later, low-level) logic for the program code, but also the narrative which will contribute to making the code easy to follow. (Typically, this narrative finds its way into comments and module prologues.) This narrative is a valuable aid to program reading (as, for example, in preparing for an inspection).

g. Because of the stepwise-refinement approach followed in TDSP, there is always something "interim" about design and code at every level except the lowest. Yet they have logical and operational completeness. (Perhaps it would be less misleading to say that at every level there is operational closure.)

### 3.3.2 Implications for Implementation of Design/Code Inspections

a. Since a program segment being developed according to TDSP will generally amount to no more than a page of code, one will not normally call together an inspection team of up to four persons to inspect so small a unit.

This is a matter of cost justification, an aspect we were not able to address at all in our precompiler experiments. Depending on project organization, proximity of team members, etc., it is possible that such an inspection could be conducted effectively at moderate cost.[1] In our own case, we decided that we would have to have enough code to justify at least a two-hour inspection. But when one chooses to let compiled code build up, the full effectiveness of formal inspections may not be able to manifest itself. This is because by the time the first code inspection is held, the design-code integrate cycle may already have been executed, and in producing operable code for each of the incremental expansions, the designer/coder is likely to have found errors. But removal of these errors, while necessary to produce running code, is not sufficient to produce correct code. From this standpoint, a formal inspection of the same segments should not be regarded as redundant.

b. Because of the cyclic implementation approach of TDSP, there is no point at which the design phase can be said to have been completed and coding can be begun. This prevents a straightforward implementation of formal design and code inspections.

A low-level design inspection is normally held only once, namely, when the level of detail attained is sufficient for coding. Thus, there is a timing conflict here between formal inspections and TDSP, for at any intermediate stage of TDSP development, there will be code as well as design to be inspected. But, according to TDSP practice, the design will always be a level ahead of the code.

---

[1] In our report on the COBOL/NLS experiment (Appendix A) we comment on the possible cost-effective use of telecommunication systems under certain conditions.

Should code and design inspections be kept separate, for logical or economic reasons? Will alternating between code and design inspections pose additional control and communication problems for the moderator and everyone else involved in the inspections? Since, as we pointed out above, the code of an $n^{th}$ level module can be said to depend for its full definition on the detail design of its subsidiary modules, there seems to be no compelling logical reason against mixing design and code inspections. Of course, inspectors would derive maximum benefit from having reviewed the design of the lower level modules prior to reviewing the code of the present module.

There may also be practical considerations which dictate the alternative to be chosen. For example, it may be easier to mix design and code inspections on a small project than on a large one -- for reasons of logic as well as logistics.

## 3.4 THE NLS/COBOL EXPERIMENT SUMMARY

### 3.4.1 Introduction

This experiment involved the formal inspection of COBOL precompiler code being written in the L-10 language. (Development of this precompiler was a deliverable on contract #F30602-76-C-0115 with RADC. See Appendix A.)

This code inspection was to be carried out via the teleprocessing facilities of the NLS (Online System). The inspectors were L-10 experts located on the west coast. The moderator and developer were located at IBM FSD, Gaithersburg, Maryland. It was anticipated that the NLS facilities (backed up by a voice-line hookup) would be used in the overview and inspection operations, with optional use reserved for the remaining operations (preparation, rework, and follow-on). The electronic mail facilities of NLS would be used to disseminate specifications and working documents to the members of the inspection team during the entire review process.

### 3.4.2 Inspection Operations

The overview operation was conducted via NLS, the author discussing via voice-line the inspection plan that had been sent out to the west coast participants through the "mail" facilities of NLS. This plan (which could be read on the CRT by all inspection participants) covered the general objectives of the inspection experiment, the use of the NLS shared-screen facilities, the five operations comprising the inspection (and the objectives of each), the roles of the participants in each of these operations, a tentative schedule, and the NLS procedures for effecting two-way communication between east and west coast.

The overview session lasted about one hour. All participants were of the opinion that it had served the intended purpose and that the NLS facilities had provided excellent support for the discussions.

No significant problems were communicated during the preparation operation.

The code inspection operation lasted 1 hour and 45 minutes. During this period, the code representing the DOWHILE and DOUNTIL process of the COBOL precompiler were inspected (about 100 lines of executable L-10 code).

Code reading proceeded very smoothly via voice-line and CRT display. Each participant also had before him a printed (hard copy) version of the code, the design specifications, and error check lists. Six major and[1] seven minor errors were found.

During the rework operation, the designer or coder resolves all errors or problems posted during the inspection. It should be mentioned that, because of the great flexibility of the NLS shared-screen facilities, the west coast inspectors showed a tendency (during the inspection operation) to demonstrate online how certain errors or problems could be resolved. Thus, some of the error correction work was actually done during the inspection -- contrary to standard formal inspection procedure.

Two additional errors -- one major and one minor -- were found during rework.

The follow-up operation presented no difficulties, and testing of the final code revealed no further errors.

### 3.4.3 Conclusions

The experiment showed that formal inspections can be conducted via a telecommunications system like NLS, when suitably augmented with audio. Elsewhere we discuss the circumstances under which these facilities may provide a welcome alternative to face-to-face inspections.

The author/designer of the precompiler judged that the inspection is likely to have cut debugging time for the inspected code in half.

In retrospect, it is interesting to observe that few communication problems arose during this experiment. During the JOVIAL inspections, communication did become a problem at times. Of course, these inspections involved both design and code and spanned a much larger period of time, making a certain amount of relearning inevitable. Also, the COBOL experiment involved only 100 lines of NLS code. Perhaps another factor may have been our realization that we were going to have to communicate via relatively limited channels and that therefore, unconsciously, we put a greater effort into the composition and compilation of inspection materials for the NLS/COBOL inspections than for the JOVIAL/JOCIT inspections.

Seven major and eight minor errors were found. Total time spent by all participants was about 20 man-hours. (This does not include some 20 hours devoted to inspection planning and composing, and transmitting the NLS messages containing the inspection materials.)

---

[1] A major error is one which would cause system failure.

The inspection rate attained on this experiment was 60 lines of executable code per hour. This is well below the published average of some 150 lines per hour. There were several reasons for this:

a. Not all participants were intimately familiar with the operation of the NLS system. In addition, system response was unusually slow during the period of the inspection.

b. As mentioned above, a certain amount of error correction advice was given during the code inspection.

c. Because of time pressure, the preprocessor code has been written virtually without comments. This probably led to more online clarification than would otherwise have been necessary.

Despite these minor problems, six of the seven major errors were found during inspection (86 percent). It took about one more hour of programmer time to attain error free execution.

## 3.5 THE JOVIAL/JOCIT EXPERIMENT

### 3.5.1 Introduction

This experiment involved high- and low-level design inspections as well as code inspections. The software product inspected was a JOVIAL/JOCIT precompiler written in COBOL. (See Appendix B.)

The JOVIAL/JOCIT precompiler was developed and implemented "top-down," in the sense that a hierarchic decomposition was produced which served as the structural guideline for stepwise implementation beginning with the top module of that structure. The first branch to be implemented (and the only one to be fully inspected) was that representing the CASE structure. The cyclic implementation technique was followed whereby a particular module is not coded until the detail design of its subsidiary modules has been completed and inspected. Design and code inspections were conducted separately. Both high- and low-level design were recorded in the Process Design Language (PDL).

The functional decomposition tree for the JOVIAL/JOCIT precompiler is shown in Figure 4.

It is important to note that, even in a top-down implementation, one does look ahead through the entire system structure to determine as early as possible which paths are especially critical or which modules are extraordinarily complex. Only then is one in a position to proceed with detail program design -- top-down or any other kind.

As we explained above, the top-down implementation method dictates that subsidiary modules must have been designed in detail before their parent module can be coded. The resulting cyclic processing is carried out again

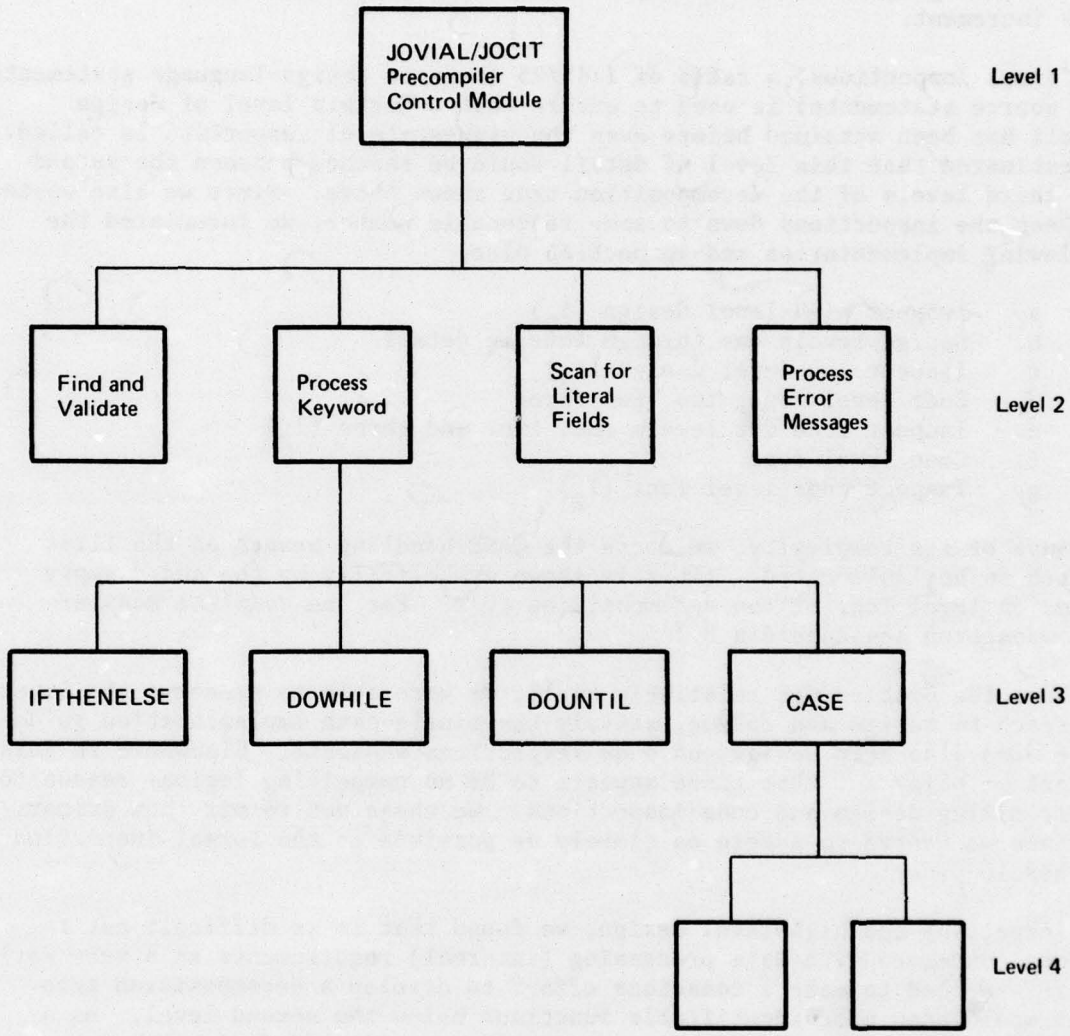



Figure 4. Functional Decomposition Tree for the JOVIAL/JOCIT Precompiler

and again as development proceeds downward through the structural decomposition tree. TDSP also suggests that design and code be expressed in small segments -- preferably small enough to fit on one page. A typical coded segment for a module on, say, level n, contains code as well as "stubs" for any calls to lower-level modules. With these stubs even a small program can be made "operable" because it is operationally closed even through it is still lacking in detail. The objective of TDSP is to have operable code at each increment.

In formal inspections, a ratio of 1:15/25 (between design-language statements and source statements) is used to ensure that a certain level of design detail has been attained before even the highest-level inspection is called. We estimated that this level of detail would be reached between the second and third levels of the decomposition tree shown above. Since we also wanted to keep the inspections down to some reasonable number, we formulated the following implementation and inspection plan:

a. Inspect high-level design ($I_0$)
b. Design levels one through four in detail
c. Inspect low-level design ($I_1$)
d. Code levels one, two, and three
e. Inspect code for levels one, two, and three ($I_2$)
f. Code level four
g. Inspect code level four ($I_2$).

Because of its complexity, we chose the CASE-handling branch as the first branch to be implemented. (This is shown symbolically by the added empty boxes on level four of the decomposition tree. For the complete modular decomposition see Appendix B.)

Because the problem was relatively small, we were able to preserve the layered approach to design and coding, satisfy the single-path implementation guideline, and also keep design and code inspections separate. Elsewhere in this report we point out that there appears to be no compelling logical reason to avoid mixing design and code inspections. We chose not to mix them primarily because we wanted to adhere as closely as possible to the formal inspection methodology.

In inspecting the high-level design, we found that it is difficult not to become concerned with data processing (internal) requirements at a very early stage. We had to make a conscious effort to develop a decomposition tree that would show user-identifiable functions below the second level. As a consequence, a number of design revisions were suggested during the inspection which, in retrospect, were seen to reflect a level of detail inappropriate for a high-level inspection. The inspection effort was not wasted, however. By the time the rework had been completed, we had not only a high-level design but a big part of the low-level design as well.

### 3.5.2 Inspection Operations

The high-level design inspection covered about 300 PDL statements, lasted two hours, and yielded six minor errors. It covered the entire precompiler design. As mentioned above, the design was too detailed for a high-level design and, as a consequence, the inspection criteria also tended to focus too much on detail. Rework, therefore, consisted mainly in adjusting the scope of the design so as to bring it closer to the ratio suggested for formal inspections (one design process statement per 15-25 source statements of code). Additional rework went into an elaboration of the design specifications, and the breaking down of the keyword algorithms into separate procedures allowing cleaner error handling.

The low-level design inspection covered some 700 PDL statements, lasted two hours, and yielded eight major and seven minor errors.[1] This inspection covered the entire design of all precompiler modules.

The code inspection of first- and second-level CASE code lasted two hours and revealed two major and five minor errors. The two major errors involved one omission and one logical error.

The lowest-level inspection of CASE handling code (represented by levels three and four on the functional decomposition tree) was completed in two sessions taking up about two and one half hours. A total of four major and five minor errors were uncovered in about 300 lines of executable COBOL code (or some 30 errors per 1,000 lines of code).

---

[1] Three of the major errors were omissions while the remaining five involved logical errors.

A summary of the JOVIAL/JOCIT inspection results is shown in Table 1.

Table 1. Summary of JOVIAL/JOCIT Inspection Results

| Formal Inspections | Errors | | Number of PDL Statements | Number of NCSS* | Major Errors per K.NCSS | Major Errors per K PDL Statements |
|---|---|---|---|---|---|---|
| | Major | Minor | | | | |
| High-Level Design Inspection ($I_0$) | -- | 6 | 300 | | | 0 |
| Low-Level Design Inspection ($I_1$) | 8 | 7 | 700 | | | 11 |
| Code Inspections ($I_0$) | 6 | 10 | | 1125 | 5.3 | |
| Unit Testing Reworked Code | 0 | 0 | | 1125 | 0 | |

*NCSS -- Noncommentary source statements; i.e., executable statements

3.5.3 Conclusions

a. Our own experience indicates that, with minor adjustments, formal inspections of high-level design, low-level design, and code can be carried out effectively within a TDSP framework.

b. The TDSP methodology focuses on control flow because practical experience seemed to indicate (and our own experience has confirmed) that most major errors are logical errors. We must add to this, however, that we experienced some difficulty in classifying errors. The logical category therefore became a receptacle for several "unclear" cases.

c. The consensus is that the checklists for the particular programming language (COBOL) and the particular inspection being conducted were quite useful in that they facilitated a systematic inspection. (For an example of inspection checklists see Figures 7 and 8.) The inspectors expressed the opinion that a cross-reference list of all program entities would have been very useful. Ideally such a list should also indicate where the entities are defined and modified. Another desirable aid, which should be kept on display throughout an inspection, would be a structural representation of the calling hierarchy.

d. The inspectors expressed the opinion that they would not have been able to work as effectively as they did without their prior training in TDSP, particularly code reading and verification methods.

It should be noted (again) that neither the NLS/COBOL nor the JOVIAL/JOCIT experiment involved straightforward implementation of formal inspections. They were experimental tasks in which the software developed served as vehicles for the formulation of a modified formal inspection procedure that would be compatible with TDSP. It was a learning experience for all involved. The task of testing the approach proposed here still remains to be done.

## 3.6 OTHER EXPERIMENTAL DATA

Additional data has been collected from two ongoing projects in the Federal Systems Division. The first of these projects is a large realtime system, presently employing approximately 300 people. Up to this time only two or three departments have combined TDSP with formal inspections, and only one moderator has received formal training. However, project management has requested additional moderator training and plans to implement formal inspections on a project-wide basis. The following summarizes their conclusions regarding the use of formal inspections in a TDSP environment.

- Design and code increments for this project are sufficiently large that $I_1$ and $I_2$ inspections are held as separate inspections.
- The effectiveness of the moderator and the inspection process is hampered if the moderator has not received formal training.
- When the moderators were asked their opinion of the effectiveness of design/code inspections, the trained moderator was quite positive, the untrained moderators were less so.
- Two groups were inspecting 100 percent of the work produced, one group was inspecting about 50 percent, based upon the criticality and complexity of the module. All programmers had at least one module inspected.
- There is a need for management commitment by recognizing that preparation for an inspection is a high priority task for the inspectors, in most cases higher than the individual coding or design tasks of those inspectors.
- For one sub-project, for which error statistics were maintained, approximately 80 percent of the total errors reported were found during inspections.

Following (Figures 5 through 8) are samples of typical forms used in the control of the inspection process on this project. They are shown here because they have proven useful in the course of several dozen design and code inspections on a "real-world" software development effort.

The second project which was reviewed was a moderately sized (approximately 35 people) data base management/process control activity. There are two points of interest here regarding formal inspections:

- Only modules deemed critical either because of their complexity or significant system dependency were inspected.

- In some cases, because of small development increments, $I_1$ and $I_2$ inspections (design and code) were combined, resulting in a combination of code, PDL, and textual material being inspected.

Design/Code Review Checklist

Moderator

☐ 1. Determine who will attend walk-through or code review. (Four to six people generally) and notify them.

   a. Design team member
   b. Technical leader
   c. Anyone whose programs interface with product
   d. Data Base Administrator (if IMS is being used)
   e. System Analysis (design inspection only)
   f. Management (design inspection only).

☐ 2. Distribute design or code review materials at least two working days prior to scheduled meeting. Materials may consist of HIPO, PDL, charts, diagrams, tables, etc.

☐ 3. Select a recorder and furnish him with the necessary forms.

☐ 4. Serve as moderator at the actual design or code review.

☐ 5. Get a copy of errors from recorder.

☐ 6. Have all errors listed been corrected?

☐ 7. Update error list in the Design/Code Inspection Report Book which is maintained by the department.

Figure 5. Design/Code Review Checklist (Moderator)

**Design/Code Review Checklist**

Recorder (If different from Moderator)

- ☐ 1. During the design or code review, keep a comprehensive list of all errors on the Structured Walk-Through/Code Review Error Form.
- ☐ 2. Give a copy of the errors listed to the programmer who will make the corrections.
- ☐ 3. Get an estimate of elapsed time to make the corrections from the programmer.
- ☐ 4. Complete the Design/Code Inspection Report and file it along with a copy of the Structured Walk-Through/Code Review Error Form containing the detected errors in the Design/Code Inspection Report Book in the CRP bookcase.
- ☐ 5. Contact the programmer to insure detected errors have been corrected after the estimated time for correction given by the programmer has elapsed.
- ☐ 6. When all errors have been corrected, initial Design/Code Review Error form signifying all errors have been corrected.

Figure 6. - Design/Code Review Checklist (Recorder)

Design/Code Inspection Error List

(1) Product Name _____ (2) Product Type _____ (3) Date __/__/__

(4) Function _____ (5) System _____ (6) Time __:__

(7) Moderator _____ (8) Recorder _____ (9) # of Participants ____

(10) Description _____

_____

_____

Initial when all errors corrected _____

| Num. | Description | Date Corrected |
|---|---|---|
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

Figure 3.3 Page ___ of ___

Figure 7. Design/Code Inspection Error List

## Design/Code Inspection Error List

(1) Product Name _____ (2) Product Type _____ (3) Date __/__/__

(4) Function _____ (5) System _____ (6) Time __:__

(7) Moderator _____ (8) Recorder _____ (9) # of Participants _____

(10) Description _____

_____

_____

(11) Problems found

| | Description | Missing | Wrong | Extra | Sub-Total | |
|---|---|---|---|---|---|---|
| a | Standards (ST) | | | | ST | |
| b | Prologue or Prose (PR) | | | | PR | |
| c | Logic (LO) | | | | LO | |
| d | Modularity (MO) | | | | MO | |
| e | Requirements Implementation (RQ) | | | | RQ | |
| f | Interface Calls (IC) | | | | IC | |
| g | Interface Requirements (IR) | | | | IR | |
| h | Passed Data Areas (PD) | | | | PD | |
| i | Syntax (SY) | | | | SY | |
| j | Data Definitions (DD) | | | | DD | |
| k | Register Usage (RU) | | | | RU | |
| l | Tests and Branches (TB) | | | | TB | |
| m | Input/Output (IO) | | | | IO | |
| n | IMS Calls (IM) | | | | IM | |
| o | SSA Definitions (SA) | | | | SA | |
| p | Return codes/messages (RM) | | | | RM | |
| q | More Detail (MD) | | | | MD | |
| r | Module Attributes (MA) | | | | MA | |
| s | Resource Utilization (UT) | | | | UT | |
| t | Readability (RE) | | | | RE | |
| u | Maintainability (MN) | | | | MN | |
| v | Performance/Efficiency (PE) | | | | PE | |
| w | Other (OT) | | | | OT | |
| x | | | | | | |
| | | | | | | |

(12) Target Date for corrections __/__/__ (13) Actual Man Hours of Inspection ___.___

(14) Estimated Rework Hours ___.___ *(15) Estimated Lines of Code Reviewed _____ (total

*(16) Number of lines changed (N/A if new) _____

*(17) Estimated Lines of Code in Rework _____

*Applicable for code review only.

Figure 8. Design/Code Inspection Summary

## 4.1 GENERAL

The following observations and recommendations can be made on the basis of our review of current technology and our own experience with formal inspections on the precompiler development projects.

a. RADC should introduce formal inspections on a variety of operational software development projects. Our own project was not a straightforward application of formal inspections. We had to shape a compromise methodology as we went along. Our sample was far too small and the whole development effort too difficult to control adequately to serve as a basis for valid statistical testing.

b. In implementing formal inspections, RADC should insist on following the guidelines provided in the documentation developed by M. Fagan and his associates. This should include the use of available checklists and the development of new ones for use with high-level languages of special interest to RADC. Inspection criteria should be augmented in accordance with the particular high-level design methodology selected by RADC.

Various systematic approaches to design have been proposed and implemented in the last few years. To our knowledge, none of these approaches covers the full range of design problems (refer to Appendix C). Some emphasize control flow, others stress data structures. Obviously, neither can be ignored in any design task. But the control flow approach is likely to be more suitable for systems that are algorithm oriented, whereas the data oriented approach may lend itself more readily to the structuring of systems that are heavily transaction oriented. Such transaction oriented problems are the favorite examples of design researchers. This is not surprising because, by their nature, such problems are readily transformable into data processing structures, and usually the objective is to develop a system which is function-equivalent (to an existing system) but execution enhanced.

C. A. R. Hoare[1] touched on this problem when he remarked that "in many applications, algorithm plays almost no role, and certainly presents almost no problem. The real problem is the mass of detailed requirements; and the only solution is the discovery or

[1]Hoare, C. A. R., written comments on "Characteristics Needed for a Common High Order Programming Language," September 1975 as quoted in McGowan, C. and Kelly, J., "A Review of Decomposition and Design Methodology," Infotech State of the Art Conference on Structured Design, October 1976, pp. 53-79 .

invention of general rules and abstractions which cover the many thousands of cases with as few exceptions as possible. This is what makes a program short, simple, and reliable, as compared with one which attempts to treat every case as a special case. But the discovery of such general rules and classifications demands at least the same inventive genius as that of a classificatory biologist, or the grammarian of a natural language, or even a theoretical physicist." Indeed, Harlan Mills has suggested the use of formal grammars as powerful descriptive tools for managing just such diversity.[1] The point here, however, is that different categories of problems (applications) must be considered. For some of these, recently developed design methodologies (combined with other formal systems such as formal grammars) can provide substantial support. Where applicable, these design methodologies will also provide guidelines and supplemental criteria for even more effective design inspections.

c. Implementation of high-level inspections under TDSP presents no major problems, if only because TDSP does not specially address high-level design in the formal inspection sense.

Low-level design inspections and code inspections present a minor problem of synchronization. The scope of a formal low-level design inspection is considerably greater than the typical design or code segment of TDSP. Thus, one can decide to develop a sufficient number of design or code units to warrant a formal inspection of two hours. One is then faced with the further decision of keeping design and code inspections separate or mixing them.

If separate code and design inspections are held, the process can be described level by level (as shown in Figure 9), with design and code inspections alternating as the development process proceeds downward through the hierarchic structure.

If mixing is the choice one can inspect the design of, say, the nth level subsidiary modules and the code of the parent modules (on the n-1 level). This approach makes it possible for the same inspectors who reviewed the lower level design to also inspect the higher level code. Since the design of subsidiary modules constitutes the definition of the parent module in the TDSP approach, the mixed approach seems to have the advantage of logical continuity. It is probably best suited for an implementation strategy that proceeds along one branch of the decomposition tree rather than proceeding level by level across the entire tree. If one chooses the latter implementation strategy, the quantity of either design or code materials to be reviewed at a given level is more likely to be sufficient to justify separate inspections.

---

[2] Linger, R. C., Mills, H. D., Structured Programming Theory and Practice, being published.

| | |
|---|---|
| • High-level design (in PDL)<br>(yields modular decomposition tree of n levels) | $I_0$ inspection |
| • low-level design levels 1 and 2 | $I_1$ inspection |
| • code level 1 (stub 2) | $I_2$ inspection |
| • low-level design level 3 | $I_1$ inspection |
| • code level 2 (stub 3) | $I_2$ inspection |
| • | • |
| • | • |
| • | • |
| • | • |
| • low-level design level n-1 | $I_1$ inspection |
| • code level n-2 (stub n-1) | $I_2$ inspection |
| • low-level design level n | $I_1$ inspection |
| • code levels n-1 and n | $I_2$ inspection |

Figure 9. Level-by-Level Development and Inspection Plan

Separate inspections under a single-path implementation strategy involve covering the design of various levels in one inspection session and the code of those levels in another session. Without this "lumping," there probably will not be enough code (or design) to warrant a code (or design) inspection of two hours or more. Under TDSP, however, this approach has one potential drawback deriving from the incremental implementation philosophy that underlies it. According to that philosophy, one begins with a single segment, which is refined and expanded in successive steps. In order to have operable code at each increment, however, the code developed at a given level will not just be compiled but also run. So if one allows the code of several levels of modules (along a given path) to accumulate, one can wind up inspecting the code of some segments that have already been run (and therefore partially debugged) as well as some segments (representing the modules on the cut-off level) which have only been through a "clean compile." Whether debugging followed by a formal code inspection is acceptable is probably a matter of cost-effectiveness -- an aspect we were not able to address in our experiments.

## 4.2 TEAM COMPOSITION

As proposed in the literature on formal inspections, an inspection team of four members is probably best. Such a team would consist of the reviewee, two inspectors, and a moderator. The two inspectors may take turns reading (and paraphrasing) code and design during inspections. In some cases it may be technically advantageous for the inspectors to work closely together during the preparation phase. In our own experiments, we tried this approach when, due to conflicting personal commitments, the two inspectors could not meet the scheduled inspection dates. After several postponements, it was decided that they would need a rather thorough review of the programs and would, therefore, spend the better part of a day together, preparing for the inspection. The inspection was held at the end of that day and proceeded unusually smoothly. Both inspectors reported that they were able to work much more efficiently in this manner and were personally convinced that they had done a thorough review. Test results seem to bear them out.

The role of the moderator is a very important one in formal inspections. It is his responsibility to plan, prepare, and conduct the inspections and to make sure that errors detected are indeed removed from the system.

His principal concern during an inspection is to "keep things moving" by focusing attention strictly on error detection (and not correction), by concentrating on real problems and side-stepping trivia, and by somehow preserving the frequently sensitive balance between inquiry and inquisition.

## 4.3 TRAINING REQUIREMENTS

Training in two categories is required:

a. Training for programmers, analysts, and managers in top-down structured programming techniques, especially those that directly support the objectives of formal inspection, e.g., top-down design methods, design and program verification, and code reading

b. Training for inspection moderators.

A general background in TDSP will be of great value to coders, designers, and inspectors as well as moderators. Aside from the moderator, who should undergo special moderator training, no special training is considered necessary for the other inspection team members. It is assumed that they have a good knowledge of the programming and design languages used on the particular project as well as the design of the system being produced. It is also assumed that they know and follow the pertinent precepts of structured programming since this will greatly facilitate readability and intelligibility of the design or code being inspected. Finally, it is our conviction that code reading and stepwise verification techniques, as practiced in SP are of great help to inspectors reviewing someone else's work.

Moderators will benefit from additional training designed specifically to enhance the effectiveness of formal inspections. Such training courses have been offered periodically in IBM, and the principal investigator on this project attended such a course during the early phases of the contract.

Such a course would offer training in three principal areas:

a. Studying the inspection process and the role and responsibilities of the moderator

b. Witnessing a live inspection and critiquing it

c. Conducting (participating in) a formal inspection.

Normally, these course activities occupy two-and-one-half to three days.

# Appendix A. COBOL/NLS CODE INSPECTION

## A.1 INTRODUCTION

The scope and objectives of the COBOL/NLS experiment were determined by two related RADC contracts currently being performed by FSD. One of these contracts covers a study of online system (NLS) support for Modern Programming Practices (MPP); the other is a study of design/code verification (D/C V) under Modern Programming Practices. With concurrence from RADC, it was decided that the preprocessor which was to be written on the NLS contract would serve as the code that would be inspected as part of the D/C V contract. Specifically, this plan would accomplish the following objectives:

- It would assist in the development and testing of a precompiler for processing the control logic structures of Structured Programming (SP) within a standard COBOL program.
- It would provide a "real world" opportunity for evaluating the shared-screen facilities of NLS to accomplish code inspection where the inspectors are geographically remote from the author of the code.

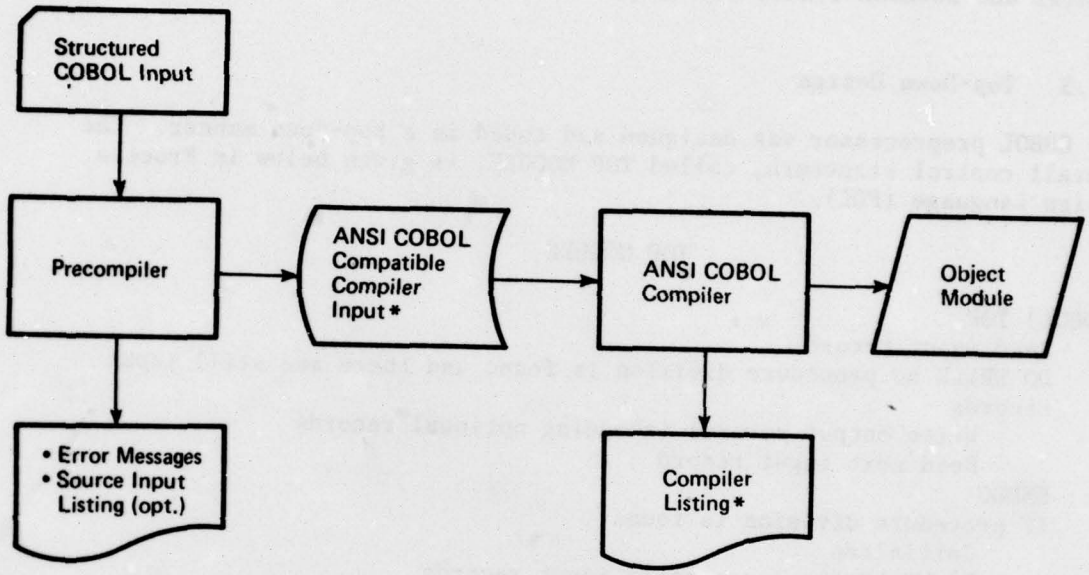
## A.2 DESCRIPTION OF THE COBOL PREPROCESSOR

### A.2.1 Purpose

A person who wants to write structured code can do so in a straightforward manner if the SP control structures translate easily into the programming language he or she is using. For example, the IFTHENELSE structure can be implemented directly in PL/I with the words IF, THEN, and ELSE.

Sometimes a logical translation has to be made first before a particular SP structure can be expressed in the language. An example of this is the conversion of a WHILEDO structure into an equivalent DO UNTIL for purposes of COBOL implementation via a PERFORM UNTIL statement.

For some languages this intermediate logical step, while possible, leads to code that no longer displays the kind of structure and readability that is the hallmark of structured programming. In that event, an additional program (a macroprocessor or a precompiler) has to be written which will accept SP control structures and translate them into equivalent statements acceptable to the standard compiler for the particular programming language (see Figure A-1). On the NLS project, the decision was made to write a precompiler that would process structured programming extensions to COBOL as a separate program execution prior to invoking the standard COBOL compiler. The precompiler itself was to be written in the L-10 language according to the standards described in Volumes I, II, and III of the Structured Programming Series, titled "Programming Language Standards," "Precompiler Specifications," and "ANSI COBOL Precompiler Program Documentation."




*Optional source input listing sequence numbers in columns 73-80.

**Figure A-1.** The Place and Function of the Precompiler in Producing an Object Module from Structured COBOL Statements

### A.2.2 Language Standards

The preprocessor is designed to accept structured COBOL (i.e., standard COBOL plus the four control structures (IFTHENELSE, DOWHILE, DOUNTIL, and CASE) written in indented format. It is designed to produce output which is acceptable to the ANSI 1974 COBOL compiler.

The L-10 code inspected on this experiment was that designed to process the DOUNTIL and DOWHILE constructs only.

### A.2.3 Top-Down Design

The COBOL preprocessor was designed and coded in a top-down manner. The overall control structure, called TOP MODULE, is given below in Process Design Language (PDL).

TOP MODULE

```
(COBOL) TOP
     Read input record
     DO WHILE no procedure division is found and there are still input
     records
          Write output records including optional records
          Read next input record
     ENDDO
     IF procedure division is found
          Initialize
          DO WHILE there are still input records
               INCLUDE process keyword VRBPROC
          ENDDO
          Complete processing of last records
     ELSE
          Error Processing
     ENDIF
     Clean up remaining processing
     STOP RUN
```

The basic design logic takes advantage of the fact that every COBOL program consists of four readily identifiable parts, called divisions:

- The IDENTIFICATION DIVISION
- The ENVIRONMENT DIVISION
- The DATA DIVISION
- The PROCEDURE DIVISION.

These divisions always occur in the order indicated above.

Since all active code will be contained in the procedure division of the COBOL program, preprocessing essentially consists in passing over all 80-column records preceding the procedure division, issuing an error message if no procedure division is found at all, and processing any procedure records

according to the control structure keywords dictated by the preprocessor format: DO, UNTIL, WHILE, ENDDO, IF, ELSE, etc.

PROCESS KEYWORD, GET RECORD, WRITE RECORD, and ERROR PROCESSING, the modules on the next lower level, are not shown in the PDL. PROCESS KEYWORD is a large CASE statement, which passes control to the proper lower program to generate the output code for DO UNTIL, DO WHILE, IFTHENELSE, and CASE figures.

The algorithms for the DOUNTIL and DOWHILE are shown below:

ALGORITHMS

DO

```
GENERATE AND STACK TWO LABELS
IF IT IS DOWHILE GENERATE BRANCH
     TO SECOND FROM TOP LABEL
GENERATE ENTRY POINT WITH TOP LABEL
SAVE CONDITION IN STACK
```

ENDDO

```
IF IT IS DOWHILE GENERATE ENTRY POINT
     WITH SECOND FROM TOP LABEL AND
     GENERATE IF CONDITION STATEMENT
IF IT IS DOUNTIL GENERATE IF
     NO (CONDITION) STATEMENT
REMOVE CONDITION FROM STACK
GENERATE BRANCH TO TOP LABEL
REMOVE TWO LABELS FROM STACK
```

Since any number of control figures of various types may be nested, proper order is maintained by continuously stacking and unstacking control indicators. In the DOWHILE/DOUNTIL figures, a "w" or "u" are stacked as flags, as is a unique label generated by a counter and literals i.e., "N---DOUNTIL" or "N---DOWHILE." These are processed by the ENDDO routine to generate the end conditions in the preprocessor output stream.

### A.2.4 Top-Down Coding

Coding and testing of the preprocessor proceeded in the order imposed by the top-down design. The TOP MODULE received the label CCOBOL. All variables used in the preprocessor are declared and initialized there.

Some functions are performed only once. These are written as INCLUDED code. Examples are verb processing (VRBPRC), which corresponds to "PROCESS KEYWORD" in the PDL design, DO processing (DOPROC), IF processing (IFPROC), etc.

Code modules that are called from many different locations are written as subroutines; e.g., error processing, get word, and condition scan. (Standard NLS subroutines were used for primitive procedures such as getting and putting records.)

The top-level code structure is shown below (Figure A-2). The inspection path is indicated by those procedures whose names appear in the darker boxes.

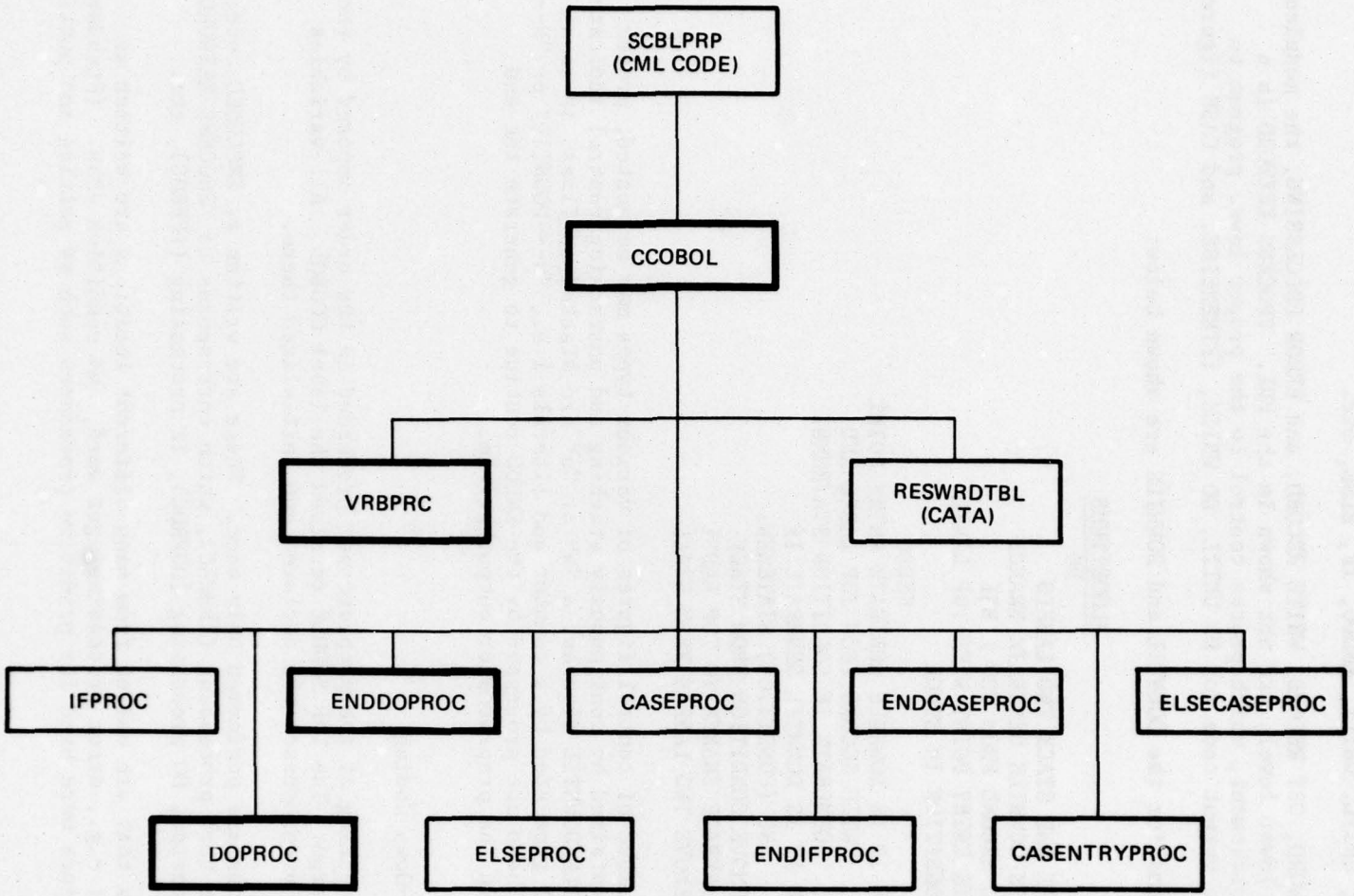



Figure A-2. Top-Down Coding of the Preprocessor

## A.3 THE COBOL/NLS CODE INSPECTION

### A.3.1 Approach

As described above, the COBOL/NLS inspection was intended to serve two purposes: to provide a test vehicle for the NLS shared-screen facility and to conduct a formal code review of the COBOL preprocessor to be written in the L-10 language. It was, therefore, decided to use NLS to provide the audio-visual communication and processing link between the west and east coast participants in this code review, which would be conducted in five phases:

a. Overview
b. Preparation
c. Inspection
d. Rework
e. Follow-up.

It was anticipated that the shared-screen facility would be used in the overview and inspection phases, with optional use reserved for the other phases. The "electronic mail" facilities of NLS would be used to disseminate specifications and working documents to the members of the inspection team during the entire review process.

### A.3.2 Overview

During this first phase of the inspection, in which all members of the inspection team participate, the designer/coder describes the overall problem area being addressed as well as the specific area he has designed (or coded) in detail. On this project the overview was conducted via NLS, the author discussing via voice-line the inspection plan that had been sent to the west coast participants through the "mail" facilities of NLS. This plan (which could be read by all participants on the CRT) covered the following topics:

a. Purpose of inspections generally, and of this inspection in particular

b. The planned use of the NLS shared-screen facilities in carrying out the inspection of the COBOL precompiler code

c. The purpose and design of the preprocessor

d. The five phases of the inspection

e. The role of the participants in each of these phases

f. The tentative inspection schedule

g. NLS procedures for effecting proper sign-on and communication between east and west coast

The west coast participants had been requested to review the program related materials that had been made available to them and to discuss them during this coast-to-coast overview.

Normally, a code inspection would have been preceded by a design inspection, which would have obviated the need for an overview during the code inspection. Since no design inspection had been held, however, it was decided that:

a. An overview operation would be conducted and would serve as a design and inspection tutorial as well as a test of the NLS equipment and facilities

b. The designer of the precompiler would serve as code reader during the actual inspection, as he would be in the best position to paraphrase the code in accordance with the design requirements.

The overview session lasted one hour, and all participants felt that the system had worked quite well and the overall objectives and individual roles were well understood. (In this overview session, as well as the subsequent inspection, the two inspectors were located in Menlo Park, California, while the moderator, author/designer, and project observer were located in Gaithersburg, Maryland.)

### A.3.3 Preparation

The preparation operation is a period during which the participants do their homework. This involves reviewing design specifications and the implementation of the design logic in the code that is to be inspected. Sometimes major programming errors are found during preparation.

On this experiment, the SRI inspectors at Menlo Park, California, were asked to note any major questions that might arise and to take them up at the beginning of the code inspection session. No significant problems arose during this operation.

The inspectors reported spending about two hours each on this preparatory operation.

### A.3.4 Inspection

The code inspection was conducted via NLS and voice-line connection between FSD, Gaithersburg and SRI, Menlo Park. The entire session lasted one hour and 45 minutes, during which time the code representing the DOWHILE and DOUNTIL processors of the COBOL precompiler was inspected. This represents about 100 lines of L-10 code.

Each participant viewed the code being read on the CRT. Additional inspection materials included hard copy of the code, the design specifications, and error checklists.

The reading of the code proceeded quite smoothly via the shared-screen facility and voice line. Particularly useful was the NLS capability that allows easy repositioning of code on the screen so as to show the full range of a structure under discussion.

Since the SRI personnel were the only L-10 experts on the inspection team, some extended discussions of the syntax and semantics of that language crept into the inspection discourse. Also, because of the great flexibility of the NLS (and the inspectors' expertise with it), there was a tendency to pursue matters of optimization and/or elegance, which are normally avoided during an inspection.

Major and minor errors were discovered in three categories (each). They are tabulated below, with references to the solutions implemented and described in the rework section below.

| TYPES | OCCURRENCES | | | SEVERITY | | Rework ERROR Reference |
|---|---|---|---|---|---|---|
| | Missing | Wrong | Extra | Major | Minor | |
| Data initialization | | 1 | | | 1 | Paragraph a |
| Data reference notation | | 2 | | 2 | | Paragraph c |
| Use of global variables | | 1 | | 1 | | Paragraph d |
| Error messages | 2 | | | | 2 | Paragraph g |
| Stacking logic | | 3 | | 3 | | Paragraph b, e |
| Efficiency | | 2 | 2 | | 4 | Paragraph f |
| Total | 2 | 9 | 2 | 6 | 7 | |

### A.3.5 Rework

In the Rework phase of a formal inspection, the designer or coder/implementer resolves all errors or problems noted during the inspection. The actions taken to make the necessary corrections to the inspected COBOL Precompiler code are summarized below.

a. Changed the initialization of the variable "cnts" from 0 to 1. This variable is used to insure uniqueness of paragraph names. While the value of zero is not an error, it is not what the programmer intended.

b. Changed the declaration of "cndstk" from an array to a string variable. This is needed in order to correct the mechanism used to stack the records holding the condition.

c. Changed "*nestk(nndx)*" to "*nestk*(nndx)" in two places.

d. The use of "wktp" in DOPROC following a call to the procedure ADDPERDC will cause malfunction because both procedures use and change "wktp." Therefore, a new text pointer called "holdtp" was declared to replace "wktp" in DOPROC.

e. In line with b. above, the change was made to store the condition in the condition stack as a string. This was done in both DOPROC and ENDDOPROC.

f. These changes were not made to correct an error but to improve the efficiency of the program by doing the same thing in a better way.

g. The deletion of "filptr-endfil" is a result of modification f. Theoretically, it is impossible to ever leave the loop in this way. The error in this case lies in the fact that if this exit did occur, there is no message to indicate that it happened.

During rework, two additional errors were found -- one major and one minor. The major one was caused by a peculiarity of the L-10 language: array references have to start at 0 whereas string references have to start at 1. It is interesting to note that this error was detected during inspection as a minor initialization error (minor because it would have caused faulty execution but no blowup). Upon reexamination during rework it became clear that this faulty initialization would, indeed, have caused a blowup.

### A.3.6 Follow-Up

During this operation, the moderator sees to it that all issues, concerns, and errors identified during inspection are resolved.

Correction of the errors was a relatively easy task as some had already been discussed during the inspection. Testing of the inspected code revealed no further errors.

### A.3.7 Evaluation of NLS Facilities

NLS (or, more appropriately, DNLS -- Display On-Line System) provided us with the capability of viewing all the documentation necessary for the review: the code review plan, the code specifications, the code design process branches, and all code developed to date. By using a partially completed command (and subsequently deleting it), each participant could place a pointer (cursor) on the other's screen. Each participant could insert statements in the file. This ability leads the participants to use the screen as a blackboard. If the file had been updated before the start of the session, both the new and old versions of the code would have been available -- and statements (even all modifications) inserted could have been subsequently deleted.

A voice telephone connection with headset or speaker-phone is an absolute necessity. The reason for this is that NLS offers no convenient way for processing system commands interspersed with typewritten messages as this would cause an inexcusable degradation of system responsiveness.

The split screen facility turned out to be impractical for this review, and was not used. First, too many documents were needed, e.g., input specs, output specs, several code modules, and the design were often needed concurrently. Further, the screen was too small to contain both significant blocks of code and fixed format output specs at the same time. Finally, the review was performed during a heavily loaded period and the use of split screen would have unduly delayed the work.

### A.3.8 Conclusions

Code inspections can be effectively conducted via NLS and voice communication. Despite inevitable delays and an occasional miscue, communication between the two inspection groups was good. The NLS shared screen facility provides very powerful and responsive communication, and participants have a tendency to go beyond error finding to error correction. Both parties can view the new code being suggested and, if there is agreement, that code can be incorporated in the current program in a matter of seconds.

Whether inspections ought to be carried out using a system such as (D)NLS is a question that can be answered only in the context of the cost, the relative geographic dispersion of a project development, and the inherent suitability of NLS for implementing the methodology of formal inspections.

Although no reliable dollar-cost estimates could be developed from just one limited experiment, a few pertinent observations can be made.

a. It is clear that a system such as NLS may well provide the only solution to the problem of obtaining occasional expert services from people located in distant parts of the country. In such a case, the capabilities of NLS (augmented by audio) can be used most effectively.

b. If one estimates the CPU cost of a time sharing system at three percent of elapsed time, the inspection itself consumed only slightly more than three minutes of CPU time.

c. The author/designer of the precompiler is of the opinion that the NLS code review is likely to have cut debugging time for the inspected code in half.

d. Overall it was felt that in this particular case the inspection was cost effective and that the cost factor may improve when highly skilled personnel from different locations must be brought together periodically, but for short periods of time, to inspect the design and code of modules constituting a much larger system than was inspected on this experiment.

e. Dissemination of pertinent inspection materials can be accomplished quite easily, rapidly, and nonredundantly via a system like NLS, especially if all pertinent documentation is routinely entered into NLS files as it becomes available. For then this dissemination amounts to no more than sending a message telling inspectors where to look in their files for the specific materials needed.

The table below contains time (in hours) estimates for people involved in the experiment.

| | Overview | Preparation | Inspection | Rework | Follow-Up |
|---|---|---|---|---|---|
| Moderator | 20+1½[1] | - | 1.75 | 1 | 2 |
| Author | 1+1[2] | - | 1.75 | 1 | 1/2 |
| Inspector 1 | 1 | 2 | 1.75 | - | - |
| Inspector 2 | 1 | 2 | 1.75 | - | - |

Past experience indicates that code inspections can be conducted at a rate of 150 lines of code per hour. The rate attained on this experiment was 60 lines of code per hour. Several reasons can be adduced for this:

a. Although the participants in the inspection had used the coast-to-coast NLS hookup for the overview operation, none of them had ever conducted an inspection via the facilities of a nationwide teleprocessing system.

b. Partly as a result of a, there were some log-on problems, partial wipeouts, as well as occasional audio difficulties.

c. Since the inspectors had served as expert consultants on the L-10 language, they not only searched for errors but also occasionally suggested more efficient ways of coding certain segments that were functionally correct. With the NLS hookup, it was easy for them to talk about such possible improvements and then to display them on the CRT in realtime.

d. At the particular time of day of the inspection (1 PM EDT), NLS turned out to be in unexpectedly heavy use and, therefore, responded quite slowly.

e. Because of time pressure, the preprocessor code had been written virtually without comments.

Despite all these minor woes, six of the seven major errors contained in the inspected code were found during the inspection, for an effectiveness rating

---

[1] Includes time devoted to composing the messages containing the inspection materials. All participants in the overview spent one hour at the NLS terminal.

[2] Includes preparation time for overview (about one hour).

of 86 percent. It took one additional hour of programmer time to attain error-free execution.

As regards methodological suitability or compatibility, there are proponents of formal inspections who see a basic conflict here. The reasons, though not yet clearly articulated, appear to center on a conviction that the intensive, person-to-person, goal-oriented communication required for effective inspections, cannot be simulated satisfactorily with telephone and CRT.

The present experiment provided little information to confirm or allay these fears, but a few observations may be made to put the results and concerns in broader perspective. The COBOL/NLS experiment involved only a small segment of code (about 100 lines) that could be reviewed in a rather confined conceptual space. Little "jumping around" through a complex design structure (such as is often required when a low-level module of a large and complex system is being inspected) was required in this case. In cases where this is required, the inevitably sequential displays of even the most responsive TP network system may begin to obscure, rather than reveal, the logical relationships among the inspected modules. And when a few people are gathered around a single display screen, there usually is not enough work space to allow them to read the screen and also make effective use of any hard copy materials needed to provide adequate reference. The larger and more complex the overall design, the more likely it is that paper shuffling will have to be reintroduced in the working environment.

The fact remains, however, that the results obtained on this experiment were most encouraging, and it is felt that additional experience and experimentation with shared-screen inspections will provide usable solutions to these potential problems.

## Appendix B. THE JOVIAL/JOCIT EXPERIMENT

## B.1 INTRODUCTION

One of the experiments initiated involved the development of a JOVIAL/ JOCIT precompiler which could serve as a test vehicle for formal design and code inspections. The JOVIAL precompiler specifications produced on this task follow the general precompiler design found in Volume 2 of the Structured Programming Series.

The precompiler was developed in top-down incremental fashion. Since the Top-Down Structured Programming (TDSP) methodology, as described in the 15-volume series, does not specifically address high-level design, it was decided to follow the FSD standard on this. This standard assumes that a design methodology is used which leads to a systematic decomposition of specified system functions into a hierarchy of modules. This hierarchic structure (called a modular decomposition tree) determines the TDSP implementation strategy. According to this strategy, implementation proceeds in cycles centering on the nodes of the decomposition tree. A node is not coded until the detailed design of its subsidiary nodes has been completed and inspected. Normally the process flows downward along major single paths. The first path to be implemented is usually chosen because it supplies a functional capability that is critical to the overall design or is of immediate value to the user.

In order to adhere as closely as possible to the basic methodology of formal inspections, we conducted design inspections separately from code inspections. A practical case can be made for mixing them. These alternatives are considered in the sections that follow.

## B.2 APPROACH

For the purpose of our integration experiment, we can distinguish three major phases in the software development process: high-level design, low-level design, and coding. High- and low-level design were recorded in PDL (Process Design Language). Coding was done in COBOL.

The high-level design ($I_0$) inspection involves a test of high-level PDL design (including the modular decompositon of the entire precompiler program) against the functional specifications. Test criteria are primarily externally (user) oriented. The level of detail is carried to a point where one PDL statement is estimated to require anywhere from 15 to 25 source statements. The level of detail for the low-level design ($I_1$) inspection should be sufficient to serve as a basis for coding. In formal inspections a ratio of one PDL statement to five to ten source statements is assumed.

The initial scope and direction of high-level design is illustrated in Figure B-1. It represents our first systematic decomposition of the JOVIAL/JOCIT precompiler development specifications into major functions. At this highest level, the decomposition criteria reflect mainly user-oriented requirements, but in a heavily data processing oriented task it is difficult to maintain that outlook very long. Internal requirements (reflecting the "how" of data processing) soon demand consideration. For example, on our task, such considerations prompted us to break down keyword processing into a number of separate modules to handle DO, IF, ELSE, CASENTRY, CASE, ELSECASE, ENDCASE,

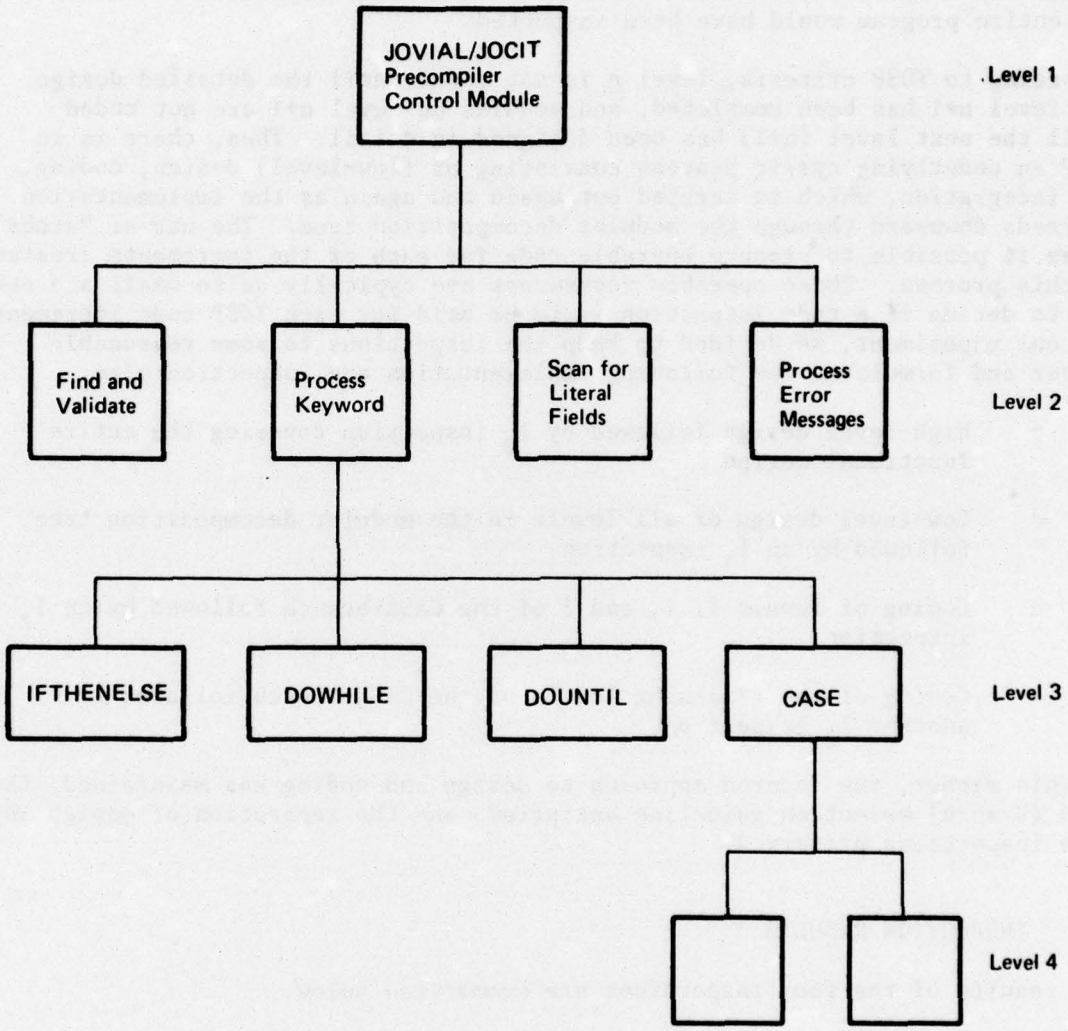



Figure B-1. User-Oriented Functional Decomposition Tree

ENDDO, and ENDIF. By refining our analysis in this way, we reached the next level of detail -- the low-level design stage. The expanded modular decomposition tree for this level is shown in Figure B-2. The area enclosed by broken lines includes the modules that were subjected to formal design and code inspections. This limitation was imposed to save project time and to accommodate the inspectors, who were "borrowed" from other departments. Otherwise the entire program would have been inspected.

According to TDSP criteria, level n is not coded until the detailed design for level n+1 has been completed, and modules on level n+1 are not coded until the next level (n+2) has been designed in detail. Thus, there is in TDSP an underlying cyclic process consisting of (low-level) design, coding, and integration, which is carried out again and again as the implementation proceeds downward through the modular decomposition tree. The use of "stubs" makes it possible to produce operable code for each of the increments created in this process. These operable increments are typically quite small and one has to decide if a code inspection is to be held for each TDSP code increment. For our experiment, we decided to keep the inspections to some reasonable number and formulated the following implementation and inspection plan:

- High-level design followed by $I_0$ inspection covering the entire functional design
- Low-level design of all levels in the modular decomposition tree, followed by an $I_1$ inspection
- Coding of levels 1, 2, and 3 of the CASE-branch followed by an $I_2$ inspection
- Coding of the remaining levels of the CASE-branch followed by another $I_2$ inspection.

In this manner, the layered approach to design and coding was maintained, the path (branch) selection guideline satisfied, and the separation of design and code inspections preserved.

## B.3 INSPECTION RESULTS

The results of the four inspections are summarized below.

### B.3.1 High-Level Design Inspection ($I_0$)

Formal inspection guidelines propose that a high-level inspection be conducted when the design has been woorked out to a level of detail where one design language statement corresponds to 15 to 25 program source statements. This is intended to be an externally (user) oriented inspection on the functional level.

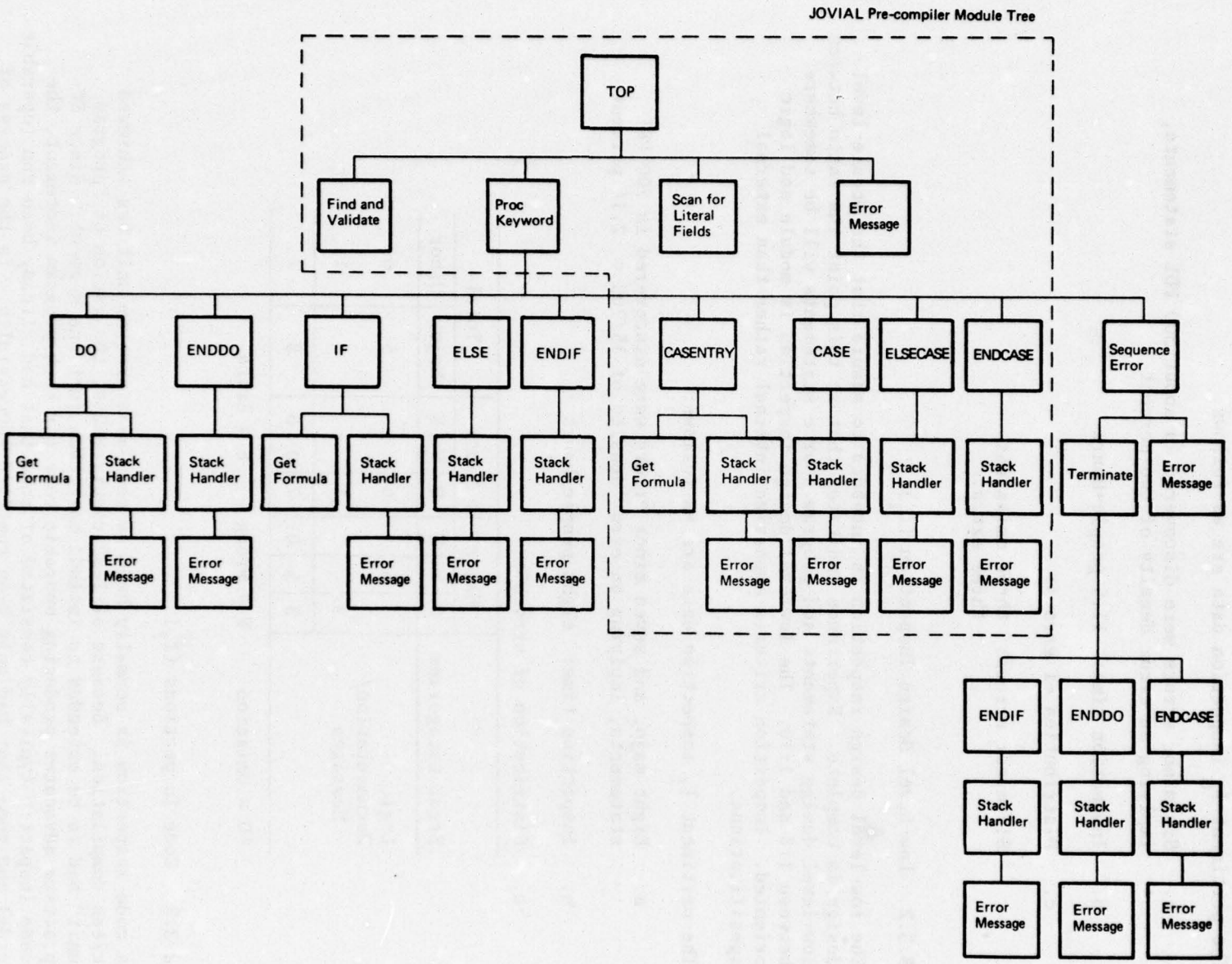



Figure B-2. Modular Decomposition Tree

The pertinent $I_0$ inspection data are as follows:

a. Six minor errors were discovered in about 300 PDL statements, implying an error density of two percent

b. Inspection time: 11.5 people-hours

c. Distribution of errors:

Six minor errors: three omissions
three extra.

### B.3.2 Low-Level Design Inspection ($I_1$)

The low-level design inspection is intended to assure that the module-level design is complete. Experience indicates that at this point the ratio between low-level design statements and program source statements will be somewhere between 1:3 and 1:10. The lowlevel design inspection is module and logic oriented. Inspection criteria emphasize internal rather than external specifications.

The pertinent $I_1$ inspection data are as follows:

a. Eight major and seven minor errors were discovered in 700 PDL statements, implying an error density of 15/700 = 2.14 percent

b. Inspection time: eight people/hours

c. Distribution of errors*:

| | Major | | | Minor | | | Total | |
|---|---|---|---|---|---|---|---|---|
| Error Categories | O | W | E | O | W | E | Major | Minor |
| Logic | | 5 | | 6 | | | 5 | 6 |
| Documentation/ Messages | 3 | | | 1 | | | 3 | 1 |
| | 3 | 5 | 0 | 7 | 0 | 0 | 8 | 7 |

*O = Omission W = Wrong E = Extra

### B.3.3 Code Inspections ($I_2$)

A code inspection is normally held as soon as a program unit has achieved clean compilation. Because of TDSP requirements, the notion of "program unit" had to be extended to include more than just one segment. Since SP practice advocates producing operable code for each program increment, the code inspected typically consisted of some that had already been run (operable code) and some that had only been compiled successfully. In the process of

developing operable code, some errors were usually found. That is to say, some error detection (and correction) had already been done before the $I_2$ inspection took place. The error density percentages given below are based solely on errors detected during the formal $I_2$ inspection.

The pertinent $I_2$ inspection data are summarized below:

First $I_2$ inspection:

a. Two major and five minor errors were discovered in 825 NCSS, implying an error density of 7/825 = one percent

b. Inspection time: eight people/hours

c. Distribution of errors:

| | Major | | | Minor | | | Total | |
|---|---|---|---|---|---|---|---|---|
| | O | W | E | O | W | E | Major | Minor |
| Logic | 1 | 1 | | | | 1 | 2 | 1 |
| Standards | | | | | 1 | 1 | | 2 |
| Error/output messages | | | | 1 | | | | 1 |
| Other | | | | 1 | | | | 1 |
| | 1 | 1 | 0 | 2 | 1 | 2 | 2 | 5 |

A detailed description of these errors (and rework action taken) can be found in Addendum B.

Second $I_2$ inspection:

a. Four major and five minor errors were detected in about 300 NCSS, implying an error density of 9/300 = three percent.

b. Inspection time: three people/hours

c. Distribution of errors:

| | Major | | | Minor | | | Total | |
|---|---|---|---|---|---|---|---|---|
| | O | W | E | O | W | E | Major | Minor |
| Logic | 1 | 3 | | | | | 4 | |
| Error/output messages | | | | 3 | 2 | | | 5 |
| | 1 | 3 | 0 | 3 | 2 | 2 | 4 | 5 |

## B.4 CONCLUSIONS

The JOVIAL/JOCIT[1] precompiler development task had two principal objectives:

a. To produce the precompiler itself, using TDSP

b. To use the growing precompiler as a test vehicle for integrating TDSP and formal inspections.

Because of the experimental approach, adaptations were made to both methodologies as we went along. Hence, the task was neither a straightforward development project nor a direct application of standard inspection techniques.

It should also be pointed out that the precompiler development task was essentially a one-person project, and the delivered version of the JOVIAL/JOCIT precompiler consisted of only about 1500 noncommented COBOL source statements. We feel that this posed a very limited test for the formal inspection methodology. Formal inspections serve to detect errors, to analyze and record them, and to use the analytical data for projection, prediction, and feedback. These aims can be realized most fully on larger software develpment projects yielding numerous measurements on individual variables. Quite clearly, the dynamics of this kind of iteration were not present on the JOVIAL precompiler development project.

Our experience indicates that it is desirable to carry out formal inspections of design and code produced according to TDSP principles. The modifications that had to be made to the two underlying methodologies were mostly directed at achieving synchronization.

High-level design is not specifically supported by TDSP and so there is no conflict of methodologies at this level. The high-level design was developed in PDL to the appropriate level of detail, whereupon the high-level inspection ($I_0$) was carried out.

Low-level design and code inspections assume an inspection unit that is typically much larger than the TDSP segment. Consequently, several increments of code and/or design will be required to make an inspection cost-justifiable. One may decide to accumulate design or code, and hold separate inspections; or one may inspect design as well as code at the same inspection. Development teams in IBM who employed the latter approach felt that inspectors who had just reviewed the design of (lower-level) subsidiary modules, were logically and actually in a better position to review the code of a particular module that they would have been if (1) they had inspected the code at some later time, or (2) if they were called upon to inspect code whose lower-level design they had not inspected at all.

---

[1]The JOVIAL/JOCIT precompiler described here was inspected and unit-tested. Its purpose was strictly experimental, however, and no attempt was made to satisfy criteria related to other environments.

Available checklists were of considerable help -- both generally (by providing direction to the inspection) and specifically (by reducing the probability of important criteria being overlooked).

Supporting documentation typically consisted of PDL and/or code, pertinent specifications, checklists for COBOL and for the particular inspection being conducted, a list of error categories, and (in most cases) a brief typewritten summary of objectives. On several occasions, the inspectors remarked that a large chart of the calling hierarchy of the entire system should be kept on display throughout the inspections. Another valuable item (not available to our inspectors) would have been a cross-reference listing of all program entities. Such a listing should, ideally, show where these items were defined and whether and where they were modified. Samples of inspection instructions, checklists, and error analyses are included as addenda to this report.

In retrospect, we are inclined to conclude that our low-level design and code inspections might have been more effective if we had mixed them instead of keeping them separate. Our second $I_2$ inspection was completed in two sessions, about a week apart. At the first session it became apparent that the inspectors, who had been away from the precompiler problem for about a month, had grown stale and had not been able to devote enough time to offset this. A reinspection was decided upon. In preparing for the reinspection, the two inspectors worked together for some five to six hours. The inspection was conducted immediately afterwards and it progressed very smoothly. Perhaps if the code inspection had been immediately preceded by a design inspection of the subsidiary modules, these problems might not have occurred -- first of all because the coverage of each inspection would have been smaller in a mixed strategy, and because the design inspection might have served as direct preparation for the (higher-level) code inspection.

## Addendum A. HIGH-LEVEL DESIGN INSPECTION CHECKLIST

**ERROR CHECKLIST**

All errors will be categorized as major or minor. They will be broken down further into the following categories:

a. Error of omission (omission)
b. Error of commission (error)
c. Error of redundancy (extra)

The principal questions to be answered during the high-level design inspection are these:

a. Is the design consistent with the JOVIAL/JOCIT precompiler specifications?

b. Does it cover all and only those specifications?

c. Is the design sufficiently detailed to serve as a basis for the low-level design?

The answers to these questions will determine whether another inspection of the high-level design should be held or whether a rework operation should be carried out. The former course will be taken if far-reaching problems are discovered during this inspection. If the problems are minor and/or isolated, the normal course will be to note these problems, rework the design, and assure that the design changes address all the problems identified during the inspection.

The following checklist is an elaboration of the major criteria given above. Further elaboration is, of course, possible and you are encouraged to formulate your own criteria and checkpoints within the scope of the specifications for this design.

a. Is the broad design understandable to you? If so, do you see any major flaws in the design?

b. Are there areas in the design that are not understandable to you? Do you see any major flaws?

c. Is the logical interrelation of parts clear to you? Do you see any major flaws?

d. Is the logic of individual algorithms clear and acceptable to you?

e. Has sufficient attention been paid to external (user oriented) requirements?

f. Have human factors been properly addressed?

g. Is the control flow clear and acceptable to you?

h. Has sufficient attention been paid to abnormal conditions or exceptions? Have they been handled adequately?

i. Have all major parameters been adequately described (without going to the byte/bit level)?

OUR MAIN OBJECTIVE IS TO CHECK FOR:

- INCOMPLETENESS
- INCONSISTENCY
- REDUNDANCY.

OPTIMIZATION IS NOT A MAJOR CONCERN AT THIS POINT.

ASSUMPTIONS

a. Structured verbs will appear as the first symbol on an input record. (Therefore, no statement labels will be permitted on the same record as a structured verb.)

b. All structured verbs which appear as the first symbol of a record will be assumed valid except if they are part of a comment, character:literal field or symbol continued from the previous input record.

## Addendum B. SUMMARY OF $I_2$ INSPECTION RESULTS

| CODE CHANGES: | |
|---|---|
| a. READ - INIT - OPTION routine should be rewritten to include an output message stating which options are taken. Errors in option card should be printed. | |
| ACTION-TAKEN: Code rewritten to conform. | EM/MINOR/OMISSION |
| b. MAIN MODULE has too many END-OF-FILE checks. | |
| ACTION-TAKEN: One has been eliminated. | LO/MINOR/EXTRA |
| c. PROC-STATUS-SCAN. COL-PTR should be incremented before ENDDO (Line 122). | |
| ACTION-TAKEN: Statement added to correct problem. | LO/MAJOR/OMISSION |
| d. PROC-STATUS-SCAN. In counting nested subscript levels, if $ is found outside of $) or ($ context, the subscript level should be reset to zero (Line 99ff). | |
| ACTION-TAKEN: Code rewritten to conform. | OT*/MINOR/CHANGE |
| e. MAIN MODULE. Indent DO WHILE (Line 18). | |
| ACTION-TAKEN: Line indented. | ST/MINOR/WRONG |
| f. MAIN MODULE Line 43. If test can be replaced with a simpler condition variable. | |
| ACTION-TAKEN: New condition variable introduced. | ST/MINOR/EXTRA |
| g. MAIN MODULE. Setting of PROG-END does not set END-OF-FILE-C. Hence, an endless loop may result. | |
| ACTION-TAKEN: PROG-END is reset following the inner DO. | LO/MAJOR/WRONG |

*Reinterpretation of Specs.

SPECIFICATION UPDATES:

a. Keyword splitting at end of input record will not be allowed.

b. Comment fields on same card as structured verb will be allowed. Other code will be discarded and an error message given.

c. Preprocessor must be able to handle batched JOVIAL programs. (Present code already provides this capability.)

d. If, in processing nested subscript levels, the preprocessor encounters a dollar sign which is not preceded by a left parenthesis or followed by a right parentheses, then the subscript level count will be reset immediately to one, and the dollar sign will be assumed to be a sentence delimiter.

ERROR CODES:

| | |
|---|---|
| CONTROL BLOCK DEFINITION | CD |
| CONTROL BLOCK USAGE | CU |
| LOGIC | LO |
| DOCUMENTATION/COMMENTS/MESSAGES | DO |
| TEST & BRANCH | TB |
| STANDARDS/STYLE | ST |
| REGISTER USAGE | RU |
| PROLOGUE | PR |
| STORAGE USAGE | SU |
| DESIGN ERROR | DE |
| MAINTAINABILITY | MN |
| EXTERNAL LINKAGES | EL |
| ERROR/OUTPUT MESSAGES | EM |
| OTHER | OT |

## Appendix C. NOTES ON DESIGN AND ITS PLACE IN THE SOFTWARE DEVELOPMENT PROCESS

When we examine the subtasks and phases that constitute the software development process in order to clarify the role and scope of design, we find less than total agreement among writers on this subject.

M. Fagan[1] distinguishes 9 levels of process operations in the programming process:

| | | |
|---|---|---|
| Level 0 | Statement of Objectives | |
| Level 1 | Architecture | |
| Level 2 | External Specifications | Design |
| Level 3 | Internal Specifications | |
| Level 4 | Logic Specifications | |
| Level 5 | Coding/Implementation | Code |
| Level 6 | Function Test | |
| Level 7 | Component Test | Test |
| Level 8 | System Test | |

DoD agencies characterize the software life cycle as follows:[2]

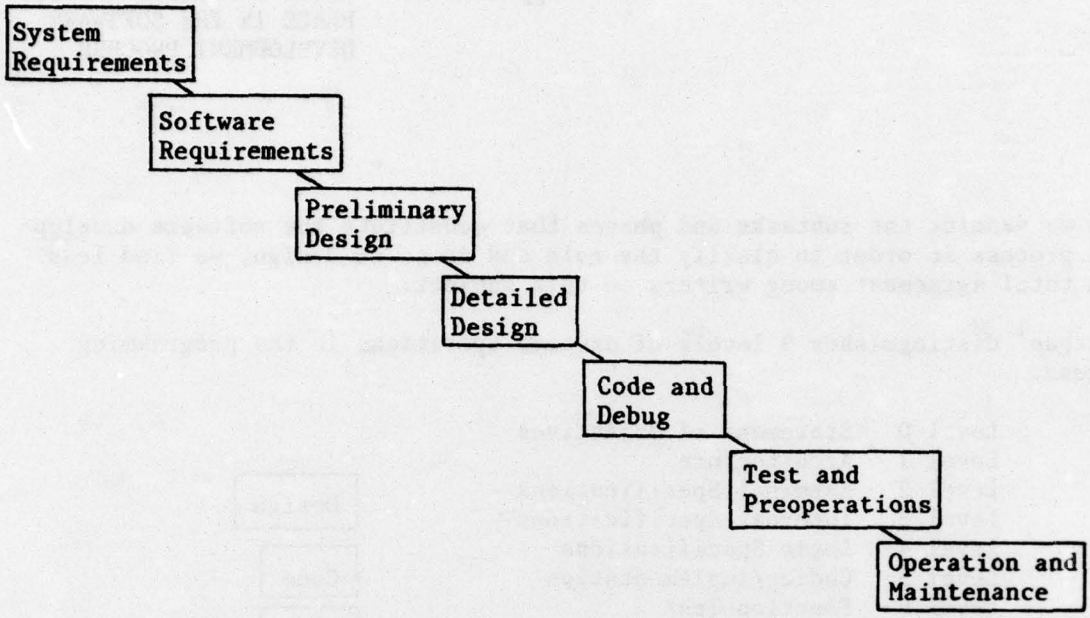
| | | | |
|---|---|---|---|
| Concept Formulation | Requirements Specification | Software Development | Operation and Maintenance |

---

[1]Fagan, M. "Design and Code Inspections to Reduce Errors in Program Development," IBM Systems Journal, Vol. 15, No. 3, 1976, pp. 182-211.

[2]Wegner, P, "Programming Languages," IEEE Transactions on Computers, Vol. C-25, No. 12, December 1976, pp. 1207ff.

Boehm[1] represents the software life cycle as follows:




J. M. Yohe[2] provides the following subdivision of the programming process:

a. Problem definition
b. Selection of algorithms and data structures
c. Selection of a programming language or languages
d. Specification of program logic and structure
e. Coding
f. Debugging and testing
g. Redefinition of above steps as necessary
h. Documentation
i. Maintenance.

Yohe remarks that problem definition is often done inadequately for two reasons:

a. The individual who originates the problem may not have a thorough understanding of it

b. Lack of communication (sometimes attributable to a lack of common background).

---

[1]Boehm, B. W., "Software Engineering, "IEEE Transactions on Computers," Vol. C-25, No. 12, December 1976, pp. 1226ff.

[2]Yohe, J. M., "An Overview of Programming Practices," Computer Surveys, Vol. 6, No. 4, December 1974.

He then proceeds to imply that problem definition is largely a matter of looking up the appropriate algorithms. Specifications, he adds, should be oriented toward the program which is to be written. (These are the internal specifications we have mentioned in the formal inspection context.)

Daniel Couger[1] distinguishes seven phases in system development.

I Documentation of the existing system

II Analysis of the system to establish requirements for an improved system (the logical design)

III Design of a computerized system (the physical design)

IV Programming and procedure development

V Implementation

VI Operation

VII Maintenance and modification.

Couger's breakdown is geared to producing what we have called function-equivalent but execution-enhanced systems. He observes, "Systems analysis, then, is concerned with Phase I and II of the system development cycle. The product of systems analysis is the logical design of the new system: the specifications for input and output of the system and the decision criteria and processing rules. Phase III, the physical design phase, determines the organization of the files and the devices to be used."

Couger also notes that systems analysis development has been trailing hardware development by a full generation. Large accounting systems presented the first large-scale application areas that were thought to justify the implementation of huge computer systems. Today, we are inclined to speculate that this may well have been a major reason for the continued limitation of the scope of analysis and design to the requirements of applications which, though large, essentially involve but an orderly implementation of a one-to-one transformation from existing manual procedures to automated ones. System specifications for such applications frequently amount to statements describing how existing operations are to be speeded up, centralized, integrated, etc. For such applications -- not always easy to implement by any means -- analysis and design can conveniently begin at a point where one can already speak concretely about files, storage requirements, etc. But this is hardly typical of application problems. In fact, Couger's is perhaps a prime example of those applications in which, according to Hoare[2], "algorithm plays almost no

---

[1]Couger, Daniel, "Evolution of Business System Analysis Techniques," Computing Surveys, Vol. 3, No. 3, 1973.

[2]Hoare, C. A. R., written comments on "Characteristics Needed for a Common High Order Programming Language," September 1975 as quoted in McGowan, C. and Kelly, J., "A Review of Decomposition and Design Methodology," Infotech State of the Art Conference on Structured Design, October 1976, pp. 53-79 .

role, and certainly presents almost no problem." He suggests that the only solution required in this type of application is the "discovery and invention of general rules and abstractions which cover the many thousands of cases with as few exceptions as possible."

A comparison of approaches to the software development process will show that:

a. There is a lack of standard terminology

b. Activities labeled the same are not likely to address the same problems

c. In some approaches, design is not mentioned specifically at all; in other cases, it appears in different places in the overall sequence (thereby implying different inputs, outputs, and exit criteria)

d. High-level design, in particular, is commonly ignored.

If we accept the dictum that the structure of the program should match the structure of the user's problem (advocated most recently by Michael Jackson), then the total design activity (high- and low-level design) must begin with the user's problem expressed in language that is familiar to him and appropriate to the nature of the problem itself. This user specification or requirement (as it is often called) must then be systematically converted into a structure that represents the user's problem completely, unambiguously, and with adequate precision. The recasting of this problem statement into a form suitable for (ultimate) processing by a computer is probably the most difficult task of software engineering today. Whatever methodology we develop for this activity must supply not only suitable form but also content. Whatever transformations we apply to the user's statement of the problem must be meaning preserving. Otherwise, we are likely to produce programs that are formally correct but do not solve the user's problem. Formal inspections can play a significant role in assuring that this does not happen. But they can do this only if high-level design can be systematized and extended in scope. This is a promising area for further exploration.

Once could begin by conducting an analysis of existing designs. Another approach would be to begin with an analysis of existing design methodologies, to investigate their scope, assumptions, degree of specificity, and extensibility, and to relate them in this fashion to the appropriate segments of the software development process as well as to different categories of application problem.

Otherwise, the nature of the problem suggests that one should look for assistance in the disciplines of formal linguistics (particularly semantics) and logic for this first task of paraphrasing English-language statements into a

form which is capable of expressing all the essential relationships implicit in the original problem statement, and which provides it own method of inference. Without this,[1] there is no way of proceeding from the "word problem" to the algorithm. Early results obtained by FSD researchers employing a semantic-analysis approach to design appear to indicate that, in principle, the method can be used to achieve program designs that compare favorably with those achieved by competent designers using more traditional methods. One additional motivating factor behind this semantics-oriented research stems from the realization that, with increasing life-cycles, the systems of the future will need to build on a base that supports not only design and implementation but also modification and ongoing communication between computer system and its technical and non-technical users.

---

[1] It is realized that, in its extreme form, the view described here could be taken to call for no less than an almost "routine" systematization of scientific discovery. The intended scope is far more modest but nonetheless indispensable. The goal is not routine scientific discovery, but the development of a systematic approach to applications-problem solving within established disciplines.

## Appendix D. GLOSSARY

| | |
|---|---|
| **Compound Program** | Any program obtained by replacing function nodes by prime programs. (As a special case, prime programs are considered compound programs themselves). |
| **Exit Criteria** | Formal criteria which must be satisfied before the current task may be considered complete. Exit criteria constitute the basis for formal inspections. |
| **Major Error** | In formal inspections, an error that causes system failure. |
| **Minor Error** | In formal inspections, an error that will adversely affect the usability or maintainability of the program. |
| **Prime Program** | A proper program which has no proper subprogram of more than one node. |
| **Process Design Language** | See Program Design Language. |
| **Program** | The combination of one or more modules to fulfill the software requirements of a project. (See Structured Program.) |
| **Program Design Language (PDL)** | An English-like language used to describe the control flow and general structure of program design. The purpose of such a language is to facilitate the systematic translation of functional specifications into data processing statements. This translation process may begin at a relatively high design level and go through several refinements before reaching the level of detail required for code writing. (Also referred to as Process Design Language). |
| **Program Segment** | A program segment is a block of code that is uniquely identifiable, especially as source code, and is a part of the logic flow of a module at execution time. Segments are organized such that their combined execution will result in the overall function of the module being performed. Usually a segment takes up a single listing page. |

| | |
|---|---|
| **Proper Program** | **A program with a control structure which**<br><br>**1. Has a single entry line and a single exit line**<br><br>**2. For each node, has a path through that node from the entry to the exit line.** |
| **Segment** | **See Program Segment.** |
| **Structured Program** | **A compound program constructed from a fixed basis set of prime programs.** |
| **Stub** | **A stub is a block of code that is temporarily used in place of a more complex block of code that is required to complete a function.** |
| **Top-Down Decomposition** | **Top-down decomposition is the activity that produces a definition of all the modules of a program, the relationships among modules (including interface specifications), a functional specification for each module, and a description of the format of any common data bases. This activity does not include the detailed design of each module.** |
| **Top-Down Structured Programming (TDSP)** | **TDSP incorporates two methodologies:**<br><br>**1. A methodology for the stepwise generation (and verification) of structured programs using a small base of prime program structures (Structured Programming)**<br><br>**2. A methodology for implementing programs in a hierarchic (top-down) sequence of cyclic processes in which design, code, integration, and testing are carried out concurrently (Top-Down Programming).** |

## Appendix E. BIBLIOGRAPHY


Alexander, C., Notes on the Synthesis of Form, Cambridge, Massachusetts: Harvard University Press, 1970.

Ascoly, J., et al., "Code Inspection Specification," IBM Technical Report 21.630, Kingston, New York, 1976.

Brooks, F. P., The Mythical Man-Month: Essays on Software Engineering, Reading, Massachusetts: Addison-Wesley, 1975.

Cammack, W. B. and Rogers, H. J., Jr., "Improving the Programming Process," IBM Technical Report 00.2483, Poughkeepsie, New York, 1973.

Chomsky, N., "Formal Properties of Grammars," in Handbook of Mathematical Psychology, Vol. 2 (R. D. Luce, R. R. Bush, and E. Galanter, eds.), New York: Wiley, 1963.

Dahl, O. J., Dijkstra, E. W., and Hoare, C. A. R., Structured Programming, New York: Academic Press, 1972.

Dijkstra, E. W., A Discipline of Programming, Englewood Cliffs, New Jersey: Prentice-Hall, 1976.

Engeler E., Introduction to the Theory of Computation, New York: Academic Press, 1973.

Estell, R. G., "Software Life Cycle Management," Installation Management Review, Vol. 5, No. 2, August 1976, pp. 2-15.

Fagan, M. E., "Design and Code Inspections to Reduce Errors in Program Development," IBM Systems Journal, Vol. 15, No. 3, pp. 182-211.

Fagan, M. E., "Design and Code Inspections and Process Control in the Development of Programs," IBM Technical Report 00.2763 (revision of IBM Technical Report 21.572), Poughkeepsie, New York, 1976.

Hopcroft, J. E. and Ullman, J. D., Formal Languages and Their Relation to Automata, Reading, Massachusetts: Addison-Wesley, 1969.

Jackson, M. A., Principles of Program Design, New York: Academic Press, 1975.

Kohli, R and Radice R. A., "Low-Level Design Inspection Specification," IBM Technical Report 21.629, Kingston, New York, 1976.

Kohli, O., "High-Level Design Inspection Specification," IBM Technical Report 21.601, Kingston, New York, 1975.

Larson, R. R., "Test Plan and Test Case Inspection Specification," IBM Technical Report 21.586, Kingston, New York, 1975.

Linger, R. C. and Mills, H. D., Structured Programming Theory and Practice (manuscript to be published by Addison-Wesley).

McCabe, T. J., "A Complexity Measure," IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, December 1976.

McGowan, C. L. and Kelly, J. R., Top-Down Structured Programming Techniques, New York: Petrocelli, 1975.

McGowan, C. and Kelly, J, "A Review of Decomposition and Design Methodology," Infotech State of the Art Conference on Structured Design, October 1976, pp. 53-73.

Mills, H. D., "Software Engineering," Science, Vol. 195, No. 4283, 18 March 1973, pp. 1199-1205.

Mills, H. D., "The New Math of Computer Programming," CACM, Vol. 18, No. 1, January 1975, pp. 43-48.

Mills H. D., "How to Write Correct Programs and Know It," Proceedings of 1975 International Conference on Reliable Software, April 21-29, 1975, Los Angeles, California.

Myers, G. J., Reliable Software through Composite Design, New York: Petrocelli, 1975.

Parnas, D. L., "On the Criteria to be Used in Decomposing Systems into Modules," CACM, December 1972, pp. 1053-1058.

Warnier, J. D., Logical Construction of Programs, New York: Van Nostrand, 1974.

Wirth, N., Algorithms + Data Structures = Programs, Englewood Cliffs, New Jersey: Prentice-Hall, 1976.

Yourdon, E. and Constantine, L., Structured Design, New York: Yourdon, Inc., 1975.