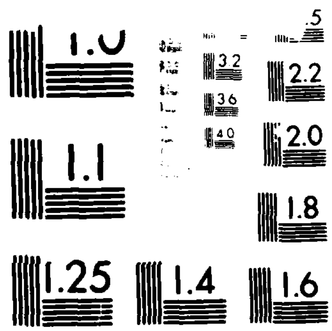
AD-A168 775 THE ROLE OF PROGRAM STRUCTURE IN SOFTWARE MAINTENANCE (U) GEORGE MASON UNIV FAIRFAX VA DEPT OF PSYCHOLOGY D A BOEHM-DAVIS ET AL. 29 MAY 86 TR-86-GMU-P01 1/2

UNCLASSIFIED N00014-85-K-0243 F/G 9/2 NL


1.0
2.5
3.2
2.2
3.6
2.0
4.0
1.1
1.8
1.25
1.4
1.6

AD-A168 775

(12)

George Mason University

TR-86-GMU-P01

THE ROLE OF PROGRAM STRUCTURE IN SOFTWARE MAINTENANCE

DEBORAH A. BOEHM-DAVIS
ROBERT W. BOLT
ALAN C. SCHULTZ
PHILIP STANLEY

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION Unclassified	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY 2b DECLASSIFICATION / DOWNGRADING SCHEDULE	3 DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited
4 PERFORMING ORGANIZATION REPORT NUMBER(S) TR-86-GMU-P01	5 MONITORING ORGANIZATION REPORT NUMBER(S) TR-86-GMU-P01
6a NAME OF PERFORMING ORGANIZATION George Mason University 6b OFFICE SYMBOL *(If applicable)*	7a NAME OF MONITORING ORGANIZATION Office of Naval Research
6c. ADDRESS *(City, State, and ZIP Code)* Psychology Department Fairfax, VA 22030	7b ADDRESS *(City, State, and ZIP Code)* Arlington, VA 22217-5000
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Engineering Psychology Program 8b OFFICE SYMBOL *(If applicable)* Code 442 1142EP	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N00014-85-K-0243
8c. ADDRESS *(City, State, and ZIP Code)* Arlington, VA 22217-5000	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO: 61153N 42 PROJECT NO: RR 04209 TASK NO: RR 0420901 WORK UNIT ACCESSION NO: NR 4424 191-0

11 TITLE *(Include Security Classification)*
The Role of Program Structure in Software Maintenance (Unclassified)

12 PERSONAL AUTHOR(S) Deborah A. Boehm-Davis, Robert W. Holt, Alan C. Schultz, Philip Stanley

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT *(Year, Month, Day)*	15 PAGE COUNT
Technical Report	FROM 85 Jan 15 TO 86 Feb 28	May 29, 1986	96

16 SUPPLEMENTARY NOTATION
Technical monitor: Dr. John J. O'Hare

17 COSATI CODES		
FIELD	GROUP	SUB-GROUP

18 SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
Software engineering, software experiments, modern programming practices, program design methodologies, software human factors, functional decomposition, Jackson program

19 ABSTRACT *(Continue on reverse if necessary and identify by block number)*
This research explores the effect of program structure on software modifiability. In this research, undergraduate computer science majors and professional programmers were asked to make either easy or difficult modifications to programs. These programs had been generated using each of three different design methodologies: in-line code, functional decomposition, and a form of object-oriented design. Further, the programmers' mental models of the structure of the programs they had studied was examined. The results suggest that problem structure, problem type, and ease of modification all affect performance. Further, they suggest that while the pattern of results is similar for professional and student programmers, the exact nature of the effect depends on the group to which the programmer belongs.

20 DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION Unclassified/unlimited
22a NAME OF RESPONSIBLE INDIVIDUAL Deborah A. Boehm-Davis	22b TELEPHONE *(Include Area Code)* (703) 323-2207 22c OFFICE SYMBOL

TR-86-GMU-P01

THE ROLE OF PROGRAM STRUCTURE IN SOFTWARE MAINTENANCE

DEBORAH A. BOEHM-DAVIS
ROBERT W. HOLT
ALAN C. SCHULTZ
PHILIP STANLEY

Psychology Department
George Mason University
4400 University Drive
Fairfax, Virginia 22030

Submitted to:

Office of Naval Research
Engineering Psychology Program
Arlington, Virginia

Contract: N00014-85-K-0243
Work Unit: NR 4424191-01

May 1986

A-1

INTRODUCTION

We have entered an era in which it has become increasingly important to develop human engineering principles which will significantly improve the structure of programs and assist programmers in ensuring system reliability and maintainability. To achieve this, it is important to understand the effects of program structure on a programmer's ability to comprehend, alter, and maintain complex programs from both a theoretical and applied perspective.

Theoretical Perspective

In order to understand the effects of program structure on programmer productivity, we must consider the way in which knowledge about computer program is cognitively represented and used by the programmer, and the way in which program structure affects the construction and use of this cognitive representation.

Cognitive representation. The basic facets of a cognitive representation or knowledge structure are the fundamental elements or entities of which the structure is composed and the relationships among those fundamental elements (Sowa, 1984). There are different views, however, on what the fundamental elements and relationships are for programmers' knowledge of computer programs.

Weiser (1982) has hypothesized that programmers cognitively deal with segments of programs that are comprised of either contiguous lines of code or of functionally related lines of code. These functional units deal with the same set of variables, forming a mini-program which Weiser calls a program "slice". Recall of programmers for debugged programs indicated that they had stored both chunks of contiguous lines of code and program slices. Thus the fundamental elements may represent either a functional unit such as a program slice or a contiguous block of code.

Adelson (1981) studied the recall of both novice and expert programmers for lines of three small computer programs. The clustered recall of the novices suggested that they were clustering lines of code from all three programs on the basis of syntactic categories such as "all IF statements". Experts, on the other hand, used the functional units of the three programs themselves to cluster their recall of the lines of code. Since these three programs contained only 16 lines of code, the size of these programs corresponded to the size of the slices discussed by Weiser.

The results for expert programmers in these two studies are consistent in indicating some functionally-based organization of the program material on the part of professional programmers. However, Adelson's results for novice programmers suggest that syntactic classification can also be used for organizing program material, and Weiser's results suggest that simple contiguity can also be used for organizing program material.

The structure organizing these basic elements of program comprehension is generally supposed to be a basic hierarchical structure of larger, more abstract elements subsuming lower-level, more detailed elements (Shneiderman & Mayer, 1979, Basili and Mills, 1982). Besides the inclusion relationship that generates a hierarchical structure, other types of relationships are possible among program chunks, such as causal relationships between a computational subroutine and an I/O subroutine that is invoked by it.

Effects of program structure. Several studies support the idea that a program with a clear, appropriate structure facilitates programmer performance. Norcio (1982) found that an indented form of documentation which clarified the control structure in a program and explained the functional nature of each program segment was superior to other forms of documentation for filling in missing statements.

Similarly, Shepard, Kruesi, and Curtis (1981) found that visually emphasizing the control flow in a program structure facilitated forward or backward tracing of the execution characteristics of the program. Boehm-Davis and Fregly (1985) found that a high-level "resource" type of documentation which emphasized the nature and structure of the communication between concurrent processes in a program facilitated modifications for this kind of complex program.

The fact that different aspects of structure emphasized in these studies facilitated programmer performance suggests that the structure emphasized by the program must be appropriate to the type of task being performed by the programmer. As Brooks (1983) stated in his discussion of a similar point, "Thus, a programmer whose task is to modify the output format will be more concerned with the output statements and less concerned with the major control structure than one who is attempting to find a bug that is causing the program to produce wrong values" (pp. 552-553). Since the above research indicates that the type of appropriate structure also varies with the inherent nature of the program, basic research studying the effects of different types of program structures across qualitatively distinct types of programs on programmer performance is necessary.

The issue of program structure has been addressed in the field of computer science in the form of program design methodologies, which seek to provide overall strategies for structuring solutions to computer problems. In general, these methods seek to improve the final program by dividing the problem into manageable parts, thus allowing the designer to deal with smaller units which are easier to code, verify, and modify. While some attempts have been made to compare specific design methodologies with each other, these comparisons have generally been non-experimental in nature and have not provided any general guidelines as to which methodologies (or which properties of methodologies) result in the most maintainable code. Such guidelines would be very useful for project managers. One approach for developing guidelines is to identify a major factor underlying the differences among methodologies and to evaluate the effect of this factor experimentally.

One fundamental difference among methodologies is the criterion used to decompose the problem into smaller units. The methodologies basically vary in the extent and type of modularization of the code. On one end of this dimension is in-line code, where all of the procedures are contained in the main routine of the program. On the other end of the dimension are techniques which rely partially on data structures and partially on operations as the basis for structuring the programs (such as object-oriented design or Parnas' information-hiding technique). Falling in between these two are techniques which rely on functions alone as the basis for structuring the problem, such as functional decomposition, or top-down design.

More specifically, in object-oriented design the criterion used to modularize the program is that one module should be created for each object (design decision) in the program. Operations are then defined for each object, and these operations are the only ones permitted on that object. In this way, each module can be created independently from the other modules in the program, i.e., does not rely on knowledge of the representation of data in any other module.

In functional decomposition the criterion used to structure the program is that each major processing step (or operation) forms one function or subroutine in the program. High-level functions or subroutines are then further decomposed into smaller ones, each of which represents a smaller processing step.

Applied Perspective

Program structure is important from an applied perspective due to the potentially large benefits that could accrue to a software project at a relatively low cost. This is true, at least in part, because improved programs reduce labor costs, especially during later phases of the software life cycle where such costs are greatest (Putnam, 1978). Recent reports have asserted that almost 70% of costs associated with software are sustained after the product is delivered (Boehm, 1981). These costs generally are spent in maintenance; that is, modifications and error corrections to the original program. These figures suggest that even small improvements in program maintainability could be translated into substantial cost savings. While many methodologies, tools, and other programming aids have been developed to produce more maintainable software, little empirical work has been done to establish either objective measures of maintainability or a particular tool's success in producing a maintainable product.

Our recent series of studies investigating the impact of documentation format on program comprehensibility, codability, verifiability, and modifiability represents a systematic, objective evaluation of the impact of a programming tool (Boehm-Davis, Sheppard, and Bailey, 1982; Sheppard, Bailey, and Bailey, 1984; Sheppard, Kruesi, and Curtis, 1981). There is, however, almost a total absence of research examining the impact of tools and methodologies early in the software process, such as in program design. Research done at TRW, IBM, and Raytheon suggests that errors made early in the project and carried on into testing and integration are the most costly type of error to find and correct. Also, characteristics of the program itself, such as its complexity, generally determine the subsequent ease of understanding and modifying the program.

Study Design

In this study, programs were created using each of three design approaches. The three program design forms were straight serial structure (in-line code), structure emphasizing functional units of the program (functional decomposition), and structure emphasizing larger object-oriented modules of the program (object-oriented). These program structures were used to write programs for each of three problems. The problems involved a real-time response system, a database system with files, and a program construcuting large linked-list data structures. Ease of maintenance for these programs was examined by presenting programmers with modifications to be made to the code and measuring the amount of time required to make those modifications. The object-oriented modularization was predicted to be most compatible with the users' internal representations of the software problems posed and thus produce the best overall performance. A further expectation was that increasing structure would increase ease of modifiability. Thus, the in-line code should produce the worst performance since it does not have any structure. Both functional decomposition and object-oriented design were predicted to lead to superior performance.

These predictions are also consistent with the demands placed on the programmers. The in-line code does not provide any structure to the program; therefore, maintenance programmers will need to build a cognitive structure as they read through and try to comprehend the program. The functional decomposition will outline modules for each function and hence provide a starting structure to programmers; however, the programmers will be required to redefine and integrate these functions into the real-world specifications for the problem, which will require some additional time for program comprehension. The object-oriented code provides one module for each real-world object, or design decision, in the system. The data and functions associated with that object are already integrated in each module. This representation scheme should allow for direct translation to the specifications, and thus, should lead to maximum performance. However, a there is a possibility that the integration of both data and functions within a module may lead to enough increased complexity to offset the benefits that should accrue from increased structure. These hypotheses are tested in this research.

METHOD

Materials

Problems. Three experimental problems and one practice problem were used in this experiment. The three experimental problems involved a military address system, a host-at-sea buoy system, and a student transactions list; all were written in PASCAL.

The military address system maintained a data base of names and postal addresses. From this data base, subsets of names, addresses, and ranks could be drawn according to specified criteria and printed according to a specified format. The host-at-sea problem involved providing navigation and weather data to air and ship traffic at sea. In this problem, buoys are deployed to collect wind, temperature, and location data and they broadcast summaries of this information to passing vessels and aircraft when requested to do so. The student transactions list problem involved storing and maintaining information about students through a transaction file using the data structure of a linked list. Copies of each version of the three problems can be seen in Appendix A.

Modifications. Two modifications were constructed for each problem: a simple and a complex modification. The simple modification required changing the program in only one location in the code. The complex modification required changing the code in several locations.

Supplemental Materials. Each problem was accompanied by five types of supplemental materials: a program overview, a data dictionary, a program listing, and listings of the current and expected output from the program. The program overview contained the program requirements, a general description of the program design, and the modification to be performed for each program. Copies of the program overviews can be found in Appendix B. The data dictionary included the variable names, an English description of the variables, and the data type for each variable. The program listing was a paper printout of the Pascal code which was identical to the code presented on the CRT screen. The listings of the current and expected output provided the programmers with the current output and the output expected from a correct run of the program; this allowed them to determine where they had gone wrong if their modification to the program did not run correctly.

Design

The experimental design used in this experiment was a 3x3x2x2 design based on Winer (1971, p. 723-736). The within-subjects factors were type or problem (military address, host-at-sea, student transactions) and program structure (in-line, functional decomposition, object-oriented). Type of modification (simple, complex) and type of programmer (undergraduates, professionals) were between-subjects variables. Each programmer was assigned, via a latin square, to modify three of the nine possible combinations of problem and program design methodology. Each programmer made either three simple modifications or three complex modifications. For example, a programmer might make a simple modification to the in-line version of the military address problem, the object-oriented version of the host-at-sea buoy problem, and the functional decomposition version of the student transactions problem. The order in which the programmers were observed under each treatment condition was randomized independently for each programmer.

Participants

The participants in this experiment were 36 programmers. Eighteen of the participants were professional programmers; these participants had an average of 3.5 years of professional programming experience. Eighteen of the programmers were upper-division undergraduate computer science majors. These participants had an average of 0.2 year of professional programming experience. Programmers were solicited through advertisements and they were paid for their participation in the research. All of the programmers had previous experience with Pascal.

Procedure

Experimental sessions were conducted on an IBM PC. Initially, the participants were given a half-hour training session in which they had to solve a sample problem. The experimenter also described the procedure for using the text editor to modify the programs during this session. This initial part of the session demonstrated the compiling and program-checking sequence. The participants were first asked to enter the changes from the problem discussed during the training session. This was done to familiarize them with the operation of the experimental system and its editor.

Following the practice program, the three experimental programs were presented. An interactive data collection system recorded the participants' responses throughout the session. The system recorded each call for an editor command (e.g. ADD, CHANGE, LIST, or DELETE). From these, the overall time to modify and debug the programs was calculated by summing the times from the individual editing sessions; the number of errors made was also calculated. The time required for compiling, linking, and executing the programs was not included in these measures. The programmers were required to continue working on a program until it was completed successfully or until 1 1/2 hours had passed. They were allowed to take breaks between programs.

After successfully modifying the problems, the programmers completed a questionnaire about their previous programming experience. The information requested included detailed information on their familiarity with programming languages, operating systems, and program design methodologies. The participants were also asked about their educational background and the extent of their professional programming experience.

Following the experiment, an attempt was made to assess the programmers' mental models of all three problems. An interactive procedure was used to elicit as much of the content of the code as the programmer recalled. This procedure was loosely based on Buschke's (1977) two dimensional grid procedure and it allowed the researcher to develop a picture of the basic units the programmer used to represent the problem and the relationships among these recalled units. Both number of recalled units and number of relationships were recorded for analysis. The recalled units were further categorized as representing primarily program slices or contiguous lines of code.

RESULTS

Professional Programmer Data

Modification Time. The participants required an average of 33 minutes to modify each program. This represents the amount of time studying the program, deciding on the appropriate changes to make the modification, and using the text editor (i.e., the total time spent at the terminal less the time for compiling, linking, executing, and checking the program).

An analysis of variance showed that, overall, it took programmers less time to make an simple modification (20 minutes) than it did to make a complex modification (47 minutes), $F(1,17) = 128.16$, $p < .01$. The analysis also showed that type of problem significantly affected the amount of time required to make the modification, $F(2,24) = 9.83$, $p < .01$. Overall, the military address problem required the least amount of time (21 minutes), the student transactions list required an intermediate amount of time (37 minutes), and the host-at-sea buoy problem required the greatest amount of time (42 minutes). The main effect of problem structure was only significant using a reduced alpha level, $F(2,24) = 2.60$, $p < .10$, and it did not interact with any of the other variables. Figure 1 shows the modification times broken down by problem structure and type of problem.

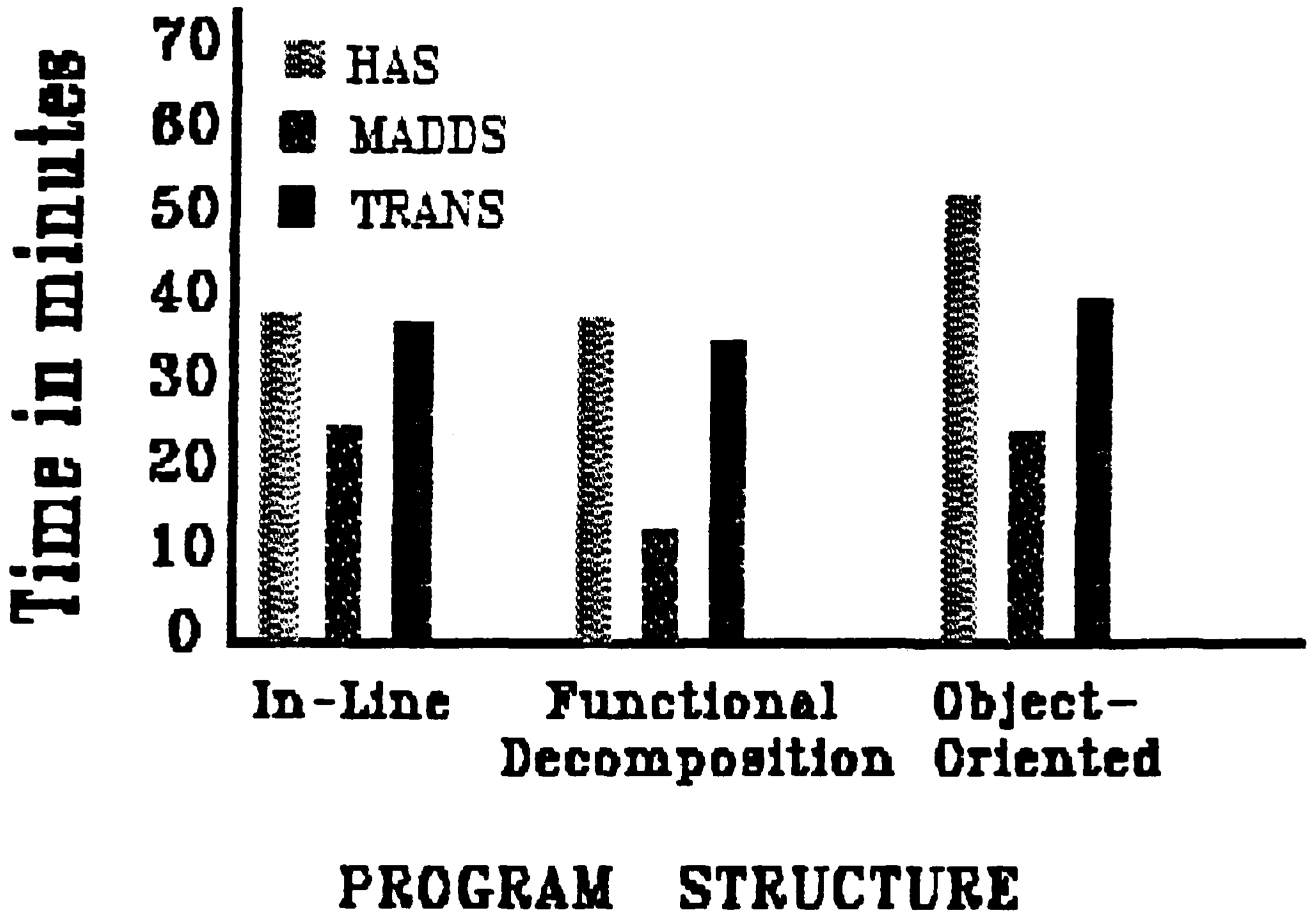


Figure 1: The interaction of program structure and problem type on time to solution for professionals.

Number of Editing Sessions. For programs that did not compile or run successfully, the programmers were required to complete another editing session. The number of sessions required to successfully modify the programs was calculated and analyzed. The analysis of variance confirmed that simple modifications required fewer sessions (1.5) than complex modifications (2.8), $F(1,17) = 9.67$, $p < .01$. No other significant results were obtained from this analysis.

Number of Editor Transactions. The number of commands executed during the editing sessions was calculated and analyzed. The analysis showed a significant main effect for type of problem ($F(2,24) = 14.07$, $p < .01$). The military address problem required the least number of transactions (14), the student transactions list required an intermediate number of transactions (37), and the host-at-sea buoy problem required the greatest number of transactions (43). In addition, the simple modifications required fewer transactions (15) than the complex modifications (47), $F(1,17) = 36.73$, $p < .01$.

Mental Models Data. The participants' mental models of the programs were assessed by asking the programmers to recall as many segments of the program as they could. They were then asked to indicate what, if any, relationships existed among the pieces they had recalled. The number of chunks recalled, and the number of relations expressed were each submitted to an analysis of variance. Both the number of chunks and the number of relations recalled were greater for the complex (4.1 and 3.1, respectively) than for the simple (3.2 and 2.0, respectively) modifications ($F(1,17) = 6.57, 12.19$, $p < .05$, respectively.

The professional programmers recalled predominantly contiguous clusters of lines of code as opposed to program slices (t (17) = 8.37, $p < .001$). The mean number of program chunks that were classified as contiguous clusters of lines of code was 9.5 while the mean number of program chunks that were categorized as program slices was 0.8.

Questionnaire Data. The post-session questionnaire contained several questions regarding the participants' programming background. The participants in this group were familiar with an average of 6.6 programming languages, 5.3 operating systems, and 2.5 program design methodologies. The questionnaire also asked them to rate (on a 7-point scale with 1 = not at all and 7 = constantly) how much they relied on each type of documentation provided. The data suggest that they relied most heavily on the program code (6.6). They relied on the program overviews (4.8), expected output (4.1) and current output (3.7) to an intermediate extent. The data dictionaries were rarely used (2.3).

Student Programmer Data

Modification Time. The student programmers required an average of 40 minutes to modify each program. An analysis of variance showed a main effect of type of modification, $F(1,17) = 19.67$, $p < .01$. The simple modifications required less time (26 minutes) than the complex modifications (54 minutes). The main effects of type of problem ($F(2,24) = 5.12$, $p < .05$) and of problem structure ($F(2,24) = 5.79$, $p < .05$) were significant. Overall, the military address problem required the least amount of time (32 minutes) while the host-at-sea buoy problem (44 minutes) and student transaction list problem (45 minutes) each required more time. Overall, the functionally decomposed code required the least amount of time (34 minutes), the in-line code required an intermediate amount of time (38 minutes) and the object-oriented code required the greatest amount of time (49 minutes). However, there were significant interactions between problem structure and type of problem ($F(2,24) = 3.44$, $p < .05$) and between type of problem and ease of modification ($F(2,24) = 5.07$, $p < .05$), so the main effect should be interpreted with caution. The nature of these interactions can be seen in Figures 2 and 3.

Number of Editing Sessions. For the student programmers, none of the independent variables significantly affected the number of editing sessions required to successfully modify the programs.

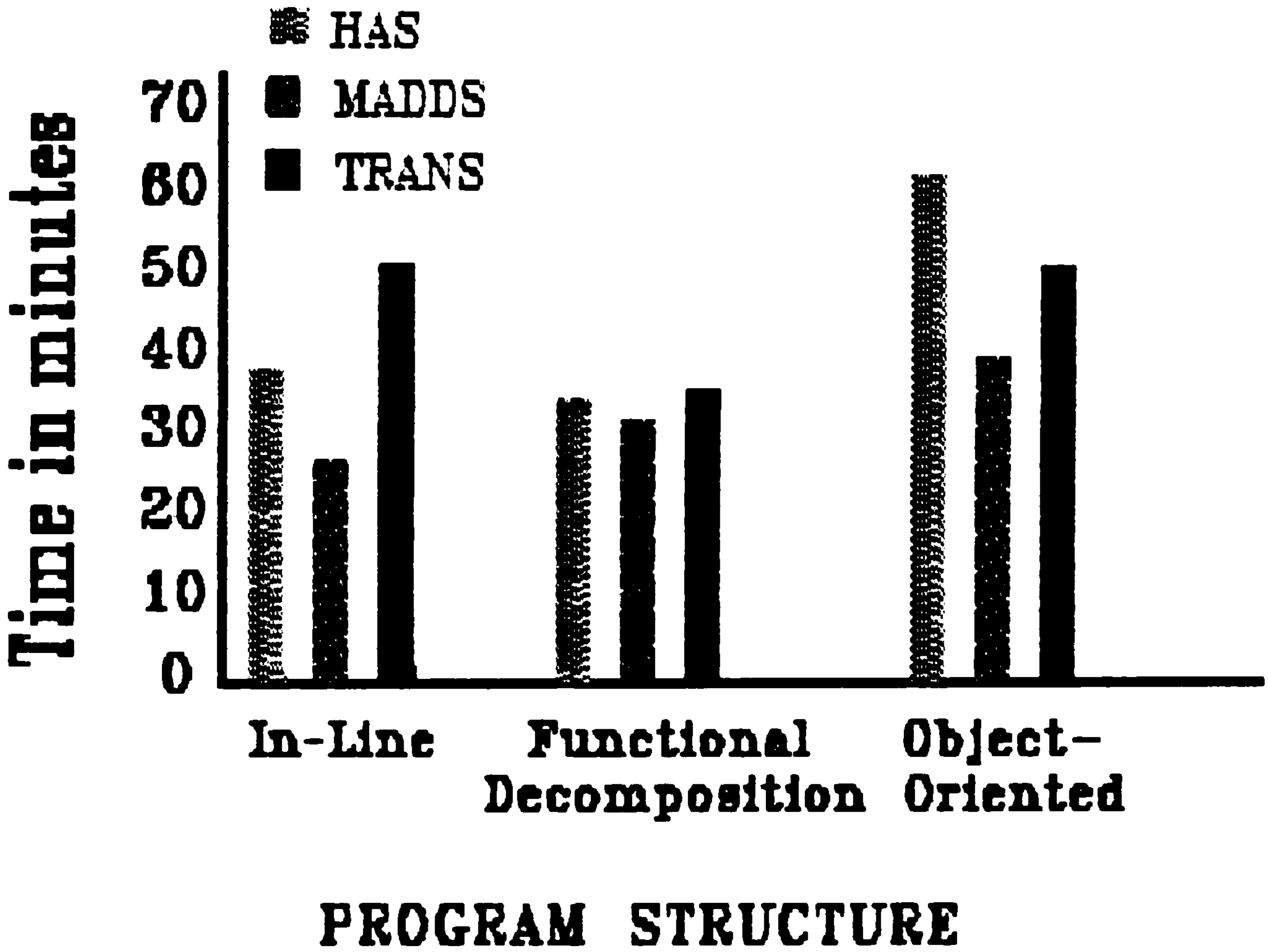


Figure 2: The interaction of program structure and problem type on time to solution for students.

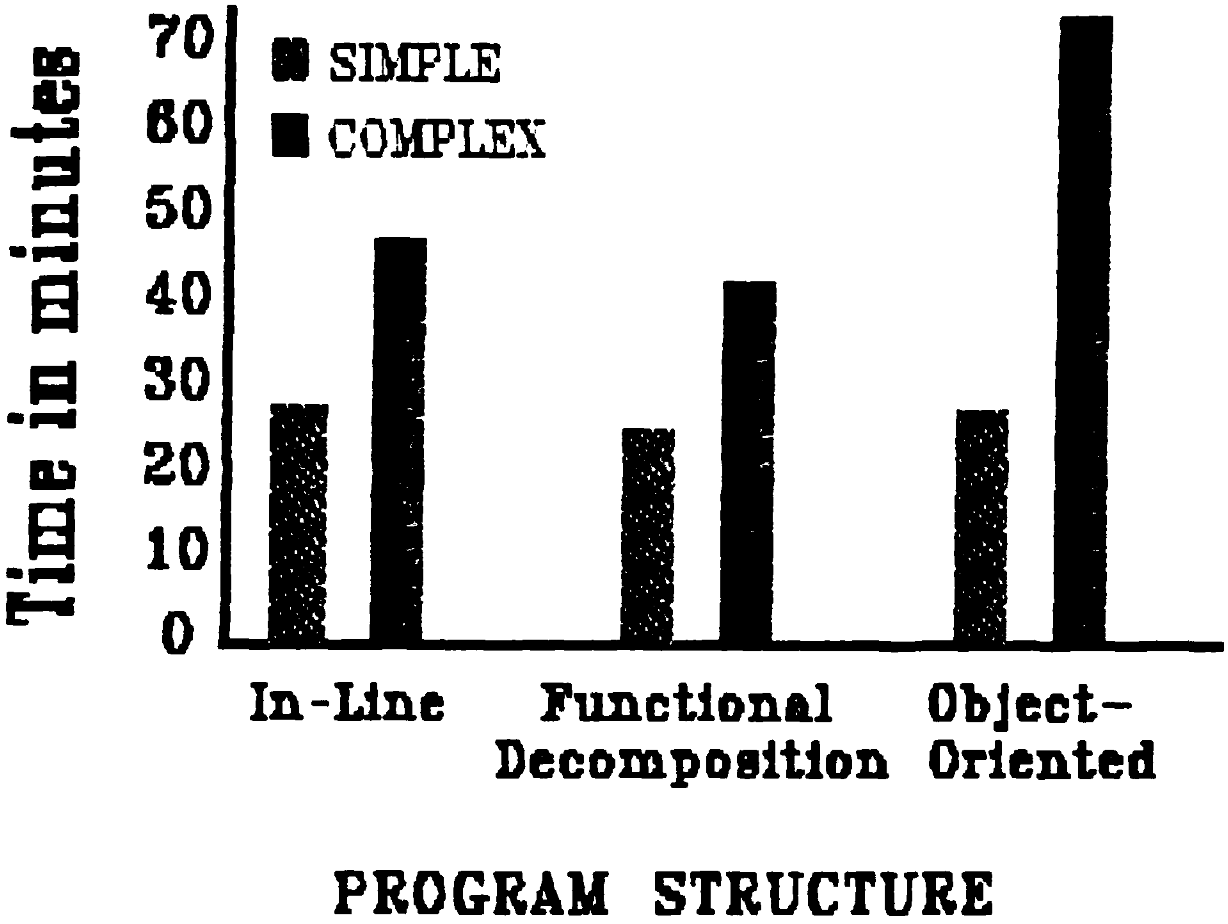


Figure 3: The interaction of program structure and type of modification on time to solution.

Number of Editor Transactions. An analysis of the number of editor transactions executed by the programmers revealed a main effect of type of modification, $F(1,17) = 11.58$, $p < .01$. The simple modifications required fewer transactions (18) than the complex modifications (35). The main effect of type of problem was also significant, $F(2,24) = 14.39$, $p < .01$. The military address problem required the smallest number of transactions (14), the host-at-sea buoy problem required an intermediate number of transactions (30) and the studen transaction list problem required the greatest number of transactions (36). In addition, there was a significant interaction between problem structure and ease of modification ($F(2,24) = 3.82$, $p < .05$). The nature of this interaction can be seen in Figure 4.

Mental Models Data. For the student programmers, the main effect of program structure was significant for both the number of chunks and relations recalled, $F(2,24) = 4.23$, 3.73, $p < .05$ for chunks and relations, respectively.

The student programmers recalled predominantly contiguous clusters of lines of code as opposed to program slices (t (17) = 5.42, $p < .001$). The mean number of program chunks recalled that were classified as contiguous clusters of lines of code was 9.6 while the mean number of program chunks that were classified as program slices was 1.3.

Questionnaire Data. The participants in this group were familiar with an average of 5.4 programming languages, 2.8 operating systems, and 2.3 program design methodologies. Of the documentation provided, the data suggest that they relied most heavily on the program code (6.0). They relied on the program overviews (5.6), expected output (4.9) and current output (4.2) to an intermediate extent. The data dictionaries were rarely used (2.6).

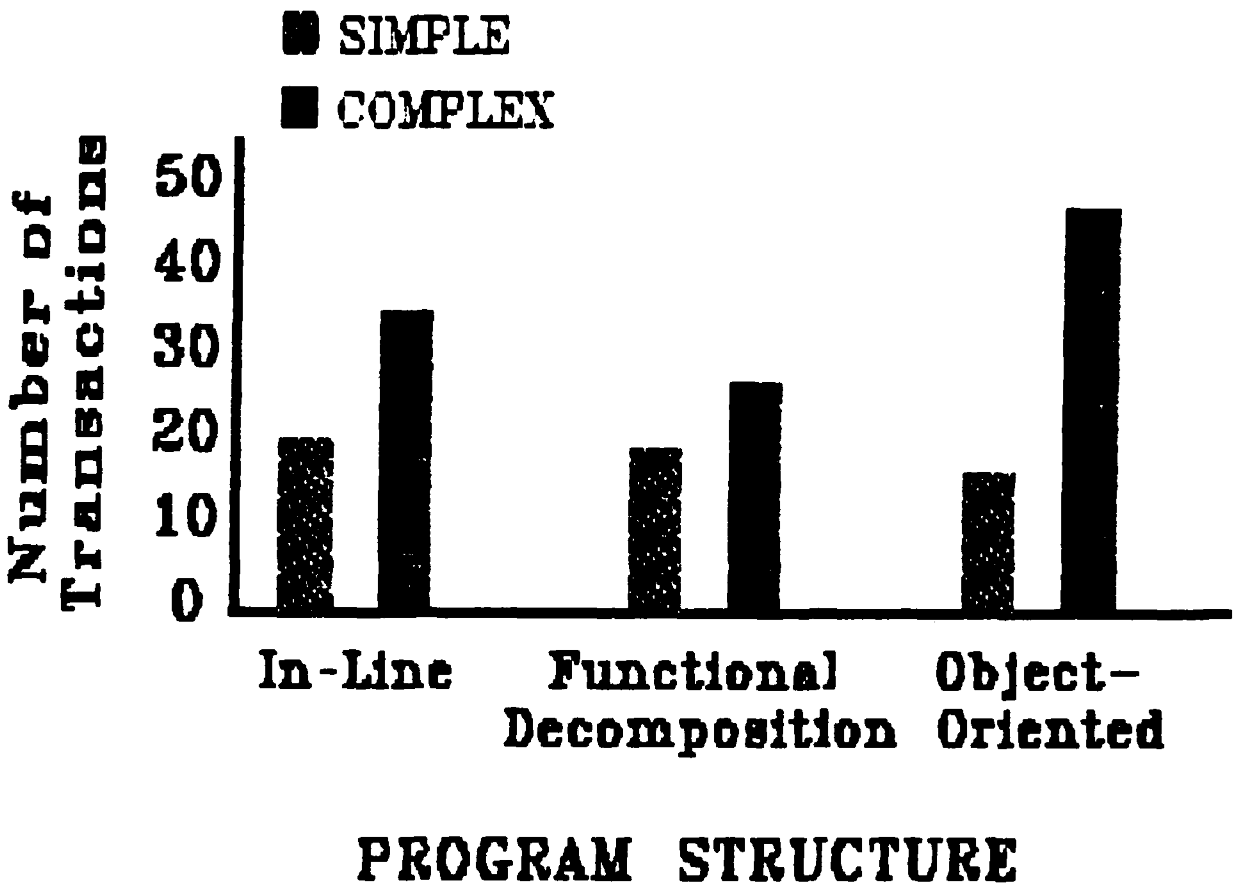


Figure 4 : The interaction of program structure and type of modification on number of editor transactions during problem solution.

DISCUSSION

The data provided by this research allow us to make several interesting observations about the role that structure plays in determining modification performance. They also provide insights into the similarities and differences between student and professional programmers.

The completion time data suggest that modification performance is influenced by an interaction between the structure of the problem and the type of problem presented. While this interaction was only statistically significant for the student programmer group, the pattern of results is very similar for the two groups of programmers. The major differences between the two groups lie in solution speed and in the effect of the object-oriented structure on the difficulty of the host-at-sea buoy problem. The professional programmers modified the military address and student transaction list problems faster than the student programmers, but modified the host-at-sea buoy problem in approximately the same amount of time as the student programmers. While the object-oriented version of the host-at-sea buoy problem required significantly more time to modify than the other versions of that problem for both groups, the effect was much more pronounced for the student programmers, leading to a significant problem structure by problem type interaction.

For both groups, substantial differences in completion time were observed between the simple and complex modifications. This difference between the types of modifications was also reflected in significant differences in the number of editor transactions for both groups of programmers and for the number of editor sessions, chunks, and relations recalled for the professional programmers. This suggests that our "complex" modifications were indeed more difficult than our "simple" modifications. This is not surprising since the complex modifications required changes in several locations of the code while our simple modifications required changes in only one location in the code.

For the student programmers, ease of modification also interacted with problem structure. This interaction revealed that for the simple modifications, problem structure did not influence ease of modification. For the complex modifications, the functionally decomposed code was easiest to modify, the in-line code was slightly more difficult to modify, and the object-oriented code was most difficult to modify. This suggests that structure, per se, is not as important as the particular type of structure.

For both groups of programmers, there was a significant difference in the completion times and number of editor transactions required to modify the three problems. In all cases, the military address problem was the easiest, while the student transaction list and host-at-sea buoy problems were roughly equal in difficulty, and more difficult than the military address problem.

The nature of the cognitive elements elicited in our free recall procedure overwhelmingly favored clusters of contiguous lines of code as opposed to program slices, as defined by Weiser (1982). Perhaps the relatively large scale of the computer programs used in this research made slicing of the computer programs too difficult, so that our programmers used the simpler strategy of clustering lines of code by continguity to form their cognitive chunks.

Differences between the student and professional programmers were found in the significance of the overall main effect of problem structure. For the professional programmers, the main effect was only significant for the time data, and only at a reduced alpha level. For the student programmers, a significant main effect was found for the time, chunk and relations data. The time data suggested that functionally decomposed code required the least amount of time, the in-line code required an intermediate amount of time, and the object-oriented code required the greatest amount of time. The number of chunks and relations recalled was lower for the in-line version of the code than for the functional decomposition and object-oriented program versions, which were equal on these measures. This suggests again that for students, structure, in and of itself, is not necessarily useful.

Overall, the data suggest that problem structure, problem type and ease of modification all affect performance. Further, the data suggest that while the pattern of results is similar for professional and student programmers, the exact nature of the effect depends on the group to which the programmer belongs. This is not surprising given the profiles of the two groups. The professionals were familiar with slightly more programming languages and operating systems while both groups were familiar with approximately the same number of program design methodologies. In addition, both groups of programmers reported relying on the same pieces of documentation, suggesting some similarities in their strategies for solving problems. The major difference between the groups was professional programming experience, with students averaging 0.2 year of experience (with a range of 0 - 1 year) while professionals averaged 3.5 years (with a range of 1.5 - 12 years).

The data, taken as a whole, only weakly supported our initial hypotheses. The data revealed that increasing program structure, as represented by our materials, did not lead to increased ease of modifiability. Overall, the functionally decomposed code was the easiest to modify, the in-line code was slightly more difficult to modify, and the object-oriented code was the most difficult to modify. An examination of the reports from the participants after they had completed the experiment suggested a trade-off between program structure and ease of modifiability. Due to the fact that the object-oriented code was the most modularized, this program structure required more passing of information from module to module. It would appear that the overhead required to keep track of the additional information is greater than the overhead reduced by the increased modularity.

In addition, the effect of program structure on modifiability was much weaker for the professional programmers than for the student programmers. The main effect of program structure was only significant for the professionals at a reduced level of confidence. One possible explanation for this result is that one skill acquired in programming professionally is the ability to adapt to many different forms of program structure.

The effects of type of problem and ease of modification were as expected. As many investigators have found, the three problems differed in their overall level of difficulty. In addition, the data strongly supported the hypothesis that changes localized in one area of the code would require less time than those modifications requiring changes in many locations in the code.

Overall, then, the data suggest that problem structure, type of problem, and ease of modification all affect modification performance for student and professional programmers, but that the exact nature of the effect depends upon the group to which the programmer belongs.

REFERENCES

Basili, V.R. and Mills, H.D. (1982). Understanding and documenting programs. IEEE Transactions on Software Engineering, SE-8(3), 270-283.

Boehm, B. W. (1981). Software Engineering Economics. Prentice-Hall, Inc.: Englewood Cliffs, N. J.

Boehm-Davis, D. A., and Fregly, A. M. (1985) Documentation of concurrent programs. Human Factors, 27, 423-432.

Boehm-Davis, D. A., Sheppard, S. B.,and Bailey, J. W. (1982). An empirical evaluation of language-tailored PDLs. In Proceedings of the 26th Annual Meeting of the Human Factors Society (pp. 984-988). Santa Monica, CA: The Human Factors Society.

Brooks, Ruven. (1983). Towards a theory of the comprehension of computer programs. Int. J. Man-Machine Studies, 18, 543-554.

Buschke, H. (1977). Two-dimensional recall: Immediate identification of clusters in episodic and semantic memory. Journal of Verbal Learning and Verbal Behavior, 12, 201-206.

Norcio, A.F. (1982). Indentation, documentation and programmer comprehension. In Proceedings of the 26th Annual Meeting of the Human Factors Society. Santa Monica, CA: The Human Factors Society, Inc.

Putnam, L. H. (1978). Measurement data to support sizing, estimating, and control of the software life cycle. In Proceedings of COMPCON '78. New York: IEEE.

Sheppard, S. B., Bailey, J. W., and Bailey, E. K. (1984). An empirical evaluation of software documentation formats. In J. C. Thomas & M. L. Schneider (Eds.), Human Factors in Computer Systems (pp. 135 - 164). Norwood, NJ: Ablex Publishing Corp.

Sheppard, S. B., Kruesi, E., & Curtis, B. (1981). The effects of symbology and spatial arrangement on the comprehension of software specifications In Proceedings of the Fifth International Conference on Software Engineering. Copyright, the Institute of electrical and Electronics Engineerrs, Inc.

Shneiderman, B., and Mayer, R. (1979) Syntactic/semantic interactions in programmer behavior: A model and experimental results. International Journal of Computer and Information Sciences, 7, 219-239.

Sowa, J.F. (1984). Conceptual Structures: Information Processing in Mind and Machine. Addison-Wesley Publishing Company, Reading MA.

Weiser, M. (1982). Programmers use slices when debugging. Communications of the ACM, 25, 446-452.

Winer, B. J. (1971). Statistical principles in experimental design. New York: McGraw-Hill.

APPENDIX A
PROGRAM CODE

Host-At-Sea Buoy Problem
(Functional Decomposition)

```
PROGRAM has (Receiver, Transmitter);     {$debug+}   {$linesize:132}
{}
{}
{}
          CONST
                Number_temp_sensors = 2;
                Number_to_avg = 5;
{}
          TYPE
                Storage_Stack =
                          RECORD
                             Top : 0..1000;
                             Data : ARRAY [1..1000] OF INTEGER;
                          END;
{}
                I_O_Type = TEXT;
{}
                Request_type = (None, Sos, Sosoff, Air, Ship);
{}
                Trans_speed_type = (Fast, Slow);
{}
{}
          VAR
                F : TEXT; {  Do NOT alter this line }
                Seconds : INTEGER;
                Transmitter : I_O_Type;
                Receiver : I_O_Type;
                Temp_gauge_1, Temp_gauge_2, Wind_s_gauge,
                Wind_d_gauge, Omega_detect : I_O_Type;
                Transmitter_speed : Trans_speed_type;
{}
                Current_request : Request_type;
                Temp1, Temp2, Omega, Wind_speed, Wind_dir : INTEGER;
                Set_sos : BOOLEAN;
                Stack : Storage_stack;
{}
{}
{}
          PROCEDURE Start_sensors;
                BEGIN
                   ASSIGN (Temp_gauge_1,  tempm1.in );
                   RESET (Temp_gauge_1);
                   ASSIGN (Temp_gauge_2,  temp2m2.in );
                   RESET (Temp_gauge_2);
                   ASSIGN (Wind_s_gauge,  windsm1.in );
                   RESET (Wind_s_gauge);
                   ASSIGN (Wind_d_gauge,  windd.in );
                   RESET (Wind_d_gauge);
                   ASSIGN (Omega_detect,  omega.in );
                   RESET (Omega_Detect);
                   Seconds := 1;
                END;
{}
{}
{}
          PROCEDURE Start_transceiver;
                BEGIN
                   RESET (Receiver);
                   REWRITE (Transmitter);
                END;
{}
```

```
{}
{}
        FUNCTION Incoming_request : Request_type;
              BEGIN
                 READLN (Receiver, Incoming_request);
              END;
{}
{}
{}
        FUNCTION  Sense (VAR  Device : I_O_Type) : INTEGER;

             BEGIN
                READLN (Device, Sense);
             END;
{}
{}
{}
        PROCEDURE  Clock_increment (VAR  Secs : INTEGER);
              BEGIN
                Secs := Secs + 1;
              END;
{}
{}
{}
         PROCEDURE  Broadcast_sos;
              BEGIN
                 WRITELN (Transmitter, 'SOS');
              END;
{}
{}
{}
         PROCEDURE  Store (Info : INTEGER);
{}
                    PROCEDURE Push (Info : INTEGER);
                         BEGIN
                            WITH Stack DO
                              BEGIN
                                 Top := Top + 1;
                                 Data [Top] := Info;
                              END; { with }
                         END;
{}
               BEGIN
                  Push (Info);
               END;
{}
{}
{}
         PROCEDURE  Broadcast_info (C_Temp1, C_Temp2, C_Omega,
                                     C_Wind_speed, C_Wind_dir
                                                             : INTEGER);
{}
              BEGIN
                 WRITELN (Transmitter, C_Temp1, C_Temp2, C_Omega,
                                        C_Wind_speed, C_Wind_dir);
              END;
{}
{}
{}
          PROCEDURE  Broadcast_detail (Detail_type : Request_type);
{}
                    VAR  Info : INTEGER;
{}
{}
                     PROCEDURE Pop (VAR Info : INTEGER);
{}
```

```
                                  BEGIN
                                    WITH Stack DO
                                     BEGIN
                                        Info := Data [Top];
                                        Top := Top - 1;
                                     END;
                                   END;
{}
{}
                             FUNCTION Empty_stack : BOOLEAN;
                                  BEGIN
                                    WITH Stack DO
                                      IF Top = 0
                                         THEN Empty_stack := TRUE
                                         ELSE Empty_stack := FALSE;
                                  END;
{}
{}
                      BEGIN
                          WHILE NOT Empty_stack DO
                             BEGIN
                                IF Detail_type = Air THEN
                                   Transmitter_speed := Fast
                                 ELSE IF Detail_type = Ship THEN
                                    Transmitter_speed := Slow;
                                Pop (Info);
                                WRITELN (Transmitter, Info);
                             END;
                      END;
{}
{}
{}
{}
               PROCEDURE  Process_request (Request : Request_type);
                    BEGIN
                       CASE Request of
                             Sos : Set_sos := TRUE;
                             Air : Broadcast_detail (Request);
                            Ship : Broadcast_detail (Request);
                          Sosoff : Set_sos := FALSE;
                       END;
                    END;
{}
{}
{}
{}
BEGIN
   Start_sensors;
   Start_transceiver;
   Stack.Top := 0;
   Seconds := 1;
   Set_sos := FALSE;
{}
   FOR VAR X := 1 TO 169 DO BEGIN  { DO NOT ALTER THIS LINE }
{}
   Clock_increment (Seconds);
   Current_request := Incoming_request;
   IF Current_request = None THEN
     BEGIN
        IF (Seconds MOD 10 = 0) THEN
           BEGIN
              IF Set_sos THEN
                 Broadcast_sos;
              Temp1 := 0;
              Temp2 := 0;
              FOR [illegible]
```

```
                    BEGIN
                          Temp1 := Temp1 + Sense (Temp_gauge_1);
                          Temp2 := Temp2 + Sense (Temp_gauge_2);
                    END;
                Temp1 := Temp1 DIV Number_to_avg;
                Temp2 := Temp2 DIV Number_to_avg;
                Store (Temp1);
                Store (temp2);
                Omega := Sense (Omega_detect);
                Store (Omega);
                IF (Seconds MOD 30 = 0) THEN
                   BEGIN
                      Wind_speed := Sense (Wind_s_gauge);
                      Store (Wind_speed);
                      Wind_dir := Sense (wind_d_gauge);
                      Store (Wind_dir);
                   END;
                 IF (Seconds MOD 60 = 0) THEN
                      Broadcast_info (Temp1, Temp2, Omega, Wind_speed,
                                   Wind_dir);
             END;
        END
      ELSE
         Process request (Current request);
{}
  END; { DO NOT ALTER THIS LINE }
{}
   {  ** Do Not alter this line ** }ASSIGN(F , RUN.DF );REWRITE(F);CLOSE(F);
END.
```

Host-At-Sea Buoy Problem
(In-Line)

```
PROGRAM Has (Receiver, Transmitter);      {+debug+}  {#linesize:132}
{}
{}
{}
         CONST
              Number_temp_sensors = 2;
              Number_to_avg = 5;
{}
         TYPE
              Storage_Stack =
                        RECORD
                           Top : 0..1000;
                           Data : ARRAY [1..1000] OF INTEGER;
                        END;
{}
              I_O_Type = TEXT;
{}
              Request_type = (None, Sos, Sosoff, Air, Ship);
{}
              Trans_speed_type = (Fast, Slow);
{}
{}
         VAR
              F : TEXT; { Do NOT alter this line }
              Seconds : INTEGER;
              Transmitter : I_O_Type;
              Receiver : I_O_Type;
              Temp_gauge_1, Temp_gauge_2, Wind_s_gauge,
              Wind_d_gauge, Omega_detect : I_O_Type;
              Transmitter_speed : Trans_speed_type;
{}
              Current_request : Request_type;
              Temp1, Temp2, Omega, Wind_speed, Wind_dir : INTEGER;
              Set_sos : BOOLEAN;
              Stack : Storage_stack;
              Sense : INTEGER;
              Info : INTEGER;
{}
{}
{}
{}
BEGIN
   ASSIGN (Temp_gauge_1,  temp1.in );
   RESET (Temp_gauge_1);
   ASSIGN (Temp_gauge_2,  temp2.in );
   RESET (Temp_gauge_2);
   ASSIGN (Wind_s_gauge,  winds.in );
   RESET (Wind_s_gauge);
   ASSIGN (Wind_d_gauge,  windd.in );
   RESET (Wind_d_gauge);
   ASSIGN (Omega_detect,  omega.in );
   RESET (Omega_Detect);
   Seconds := 1;
   RESET (Receiver);
   REWRITE (Transmitter);
   Stack.Top := 0;
   Set_sos := FALSE;
{}
   FOR VAR x := 1 TO 169 DO BEGIN  { Do NOT alter this LINE
{}
   Seconds := Seconds + 1;
```

```
READLN (Receiver, Current_request);
IF Current_request = None THEN
  BEGIN
     IF (Seconds MOD 10 = 0) THEN
        BEGIN
           IF Set_sos THEN
              WRITELN (Transmitter, 'SOS');
           Temp1 := 0;
           Temp2 := 0;
           FOR VAR Num := 1 TO Number_to_avg DO
               BEGIN
                  READLN (Temp_gauge_1, Sense);
                  Temp1 := Temp1 + Sense;
                  READLN (Temp_gauge_2, Sense);
                  Temp2 := Temp2 + Sense;
               END;
           Temp1 := Temp1 DIV Number_to_avg;
           Temp2 := Temp2 DIV Number_to_avg;
           WITH Stack DO
              BEGIN
                 Top := Top + 1;
                 Data [Top] := Temp1;
              END; { with }
           WITH Stack DO
              BEGIN
                 Top := Top + 1;
                 Data [Top] := Temp2;
              END; { with }
           READLN (Omega_detect, Omega);
           WITH Stack DO
              BEGIN
                 Top := Top + 1;
                 Data [Top] := Omega;
              END; { with }
           IF (Seconds MOD 30 = 0) THEN
              BEGIN
                 READLN (Wind_s_gauge, Wind_speed);
                 WITH Stack DO
                    BEGIN
                       Top := Top + 1;
                       Data [Top] := Wind_speed;
                    END; { with }
                 READLN (Wind_d_gauge, Wind_dir);
                 WITH Stack DO
                    BEGIN
                       Top := Top + 1;
                       Data [Top] := Wind_dir;
                    END; { with }
              END;
            IF (Seconds MOD 60 = 0) THEN
                 WRITELN (Transmitter, Temp1, Temp2, Omega,
                                Wind_speed, Wind_dir);
        END;
  END
ELSE
  CASE Current_request OF
        Sos : Set_sos := TRUE;
        Air : WITH Stack DO
                 WHILE NOT (Top = 0) DO
                    BEGIN
                       Transmitter_speed := Fast;
                       Info := Data [Top];
                       Top := Top - 1;
                       WRITELN (Transmitter, Info);
                    END;
        Ship : WITH Stack DO
```

```
                              WHILE NOT (Top = 0) DO
                                BEGIN
                                  Transmitter_speed := Slow;
                                  Info := Data [Top];
                                  Top := Top - 1;
                                  WRITELN (Transmitter, Info);
                                END;
                    Sosoff : Set_sos := FALSE;
              END; { case }
    {}
      END; { DO NOT ALTER THIS LINE }
    {}
      { *** Do NOT alter this line *** }ASSIGN(F, 'RUN.OK');REWRITE(F);CLOSE(F);
    END { has }.
```

Host-At-Sea Buoy Problem
(Object-Oriented)

```
PROGRAM Has (Receiver, Transmitter);   {#debug} {#linesize:1[illegible]}
{}
{}
     TYPE I_O_Type = TEXT;
{}
{}
{}
{}
{}
{*** OBJECT Gauges *******************************}
{}
{}
{}
          CONST Number_to_avg = 5;
{}
          TYPE  Gauge_type = (Temp_1, Temp_2, Speed, Dir, Omega);
{}
          VAR [STATIC]
                 Temp_gauge_1, Temp_gauge_2, Wind_speed_gauge,
                 Wind_dir_gauge, Omega_detect  : I_O_Type;
{}
{}
{}
PROCEDURE GAUGES__start_sensors;
          BEGIN
             ASSIGN (Temp_gauge_1,  tempm2.in );
             RESET (Temp_gauge_1);
             ASSIGN (Temp_gauge_2,  templm2.in );
             RESET (Temp_gauge_2);
             ASSIGN (Wind_speed_gauge,  windsm2.in );
             RESET (Wind_speed_gauge);
             ASSIGN (Wind_dir_gauge,  windd.in );
             RESET (Wind_dir_gauge);
             ASSIGN (Omega_detect,  omega.in );
             RESET (Omega_detect);
          END;
{}
{}
          FUNCTION Get_measurement (Gauge : Gauge_type) : INTEGER;
{}
{}
                         FUNCTION Sense (VAR Device : I_O_Type) : INTEGER;
                                BEGIN
                                   READLN (Device, Sense);
                                END;
{}
{}
                          FUNCTION Avg_temp (which : Integer) : INTEGER;
{}
                                    VAR Temp : INTEGER;
{}
                                BE[illegible]
                                   Temp : [illegible];
                                   [illegible]
                                     [illegible]
                                        IF which = 1 THEN
                                           Temp := Temp +
                                                        Sense [illegible]
                                        ELSE IF which = 2 THEN
                                           Temp :=  Temp +
                                                        Sense [illegible]
```

```
                              END;
                           Avg_temp := Temp DIV Number_to_avg;
                        END;

               BEGIN
                  CASE Gauge OF
                     Temp_1 : Get_measurement := Avg_temp (1);
                     Temp_2 : Get_measurement := Avg_temp (2);
                      Speed : Get_measurement := Sense (Wind_speed_gauge);
                        Dir : Get_measurement := Sense (Wind_dir_gauge);
                      Omega : Get_measurement := Sense (Omega_detect);
                  END; {case}
               END;
{}
{}
{}
FUNCTION GAUGES__Get_temp_1 : INTEGER;
     BEGIN
        GAUGES__Get_temp_1 := Get_measurement (Temp_1);
     END;
{}
{}
FUNCTION GAUGES__Get_temp_2 : INTEGER;
     BEGIN
        GAUGES__Get_temp_2 := Get_measurement (Temp_2);
     END;
{}
{}
FUNCTION GAUGES__Get_wind_speed : INTEGER;
     BEGIN
        GAUGES__Get_wind_speed := Get_measurement [illegible];
     END;
{}
{}
FUNCTION GAUGES__Get_wind_dir : INTEGER;
     BEGIN
        GAUGES__Get_wind_dir := Get_measurement [illegible];
     END;
{}
{}
FUNCTION GAUGES__Get_omega : INTEGER;
     BEGIN
        GAUGES__Get_omega := Get_measurement [illegible];
     END;
{}
{}
{}
{*************************************************************}
{}
{}


[illegible] *****************************************}



            [illegible]
               [illegible] = RECORD
                                   [illegible] INITIAL [illegible]
                                   [illegible]
                                END;
         [illegible]
               [illegible] : Storage [illegible];
```

```
                    PROCEDURE Push (Info : INTEGER);
                       BEGIN
                          WITH Stack DO
                            BEGIN
                               Top := Top + 1;
                               Data [Top] := Info;
                            END; { with }
                       END;
{}
{}
{}
                    FUNCTION Pop : INTEGER;
                         BEGIN
                           WITH Stack DO
                            BEGIN
                               Pop := Data [Top];
                               Top := Top - 1;
                            END;
                          END;
{}
{}
PROCEDURE MEMORY__Init_memory;
        BEGIN
          Stack.Top := 0;
        END;
{}
{}
PROCEDURE  MEMORY__Store_reading (Measurement : INTEGER);
        BEGIN
           Push (Measurement);
        END;
{}
{}
FUNCTION MEMORY__Is_memory_empty : BOOLEAN;
       BEGIN
          IF Stack.Top = 0
             THEN MEMORY__Is_memory_empty := TRUE
             ELSE MEMORY__Is_memory_empty := FALSE;
       END;
{}
{}
FUNCTION MEMORY__Get_historic_reading : INTEGER;
      BEGIN
         MEMORY__Get_historic_reading := Pop;
      END;
{}
{}
{}
{****************************************************}
{}
{}
{}
{*** OBJECT Transmitter *************************}
{}
{}
      TYPE Trans_speed_type = (Fast, Slow);
{}
      VAR [STATIC]
             Transmitter : I_O_Type;
             Transmitter_speed : Trans_speed_type;
{}
{}
PROCEDURE TRANSMITTER__Start_transmitter;
        BEGIN
           REWRITE (Transmitter);
        END;
```

```
{}
{}
PROCEDURE TRANSMITTER__Broadcast_sos;
         BEGIN
            WRITELN (Transmitter,  SOS );
         END;
{}
{}
PROCEDURE TRANSMITTER__Broadcast_info;
        BEGIN
           WRITELN (Transmitter, GAUGES__Get_temp_1,
                            GAUGES__Get_temp_2, GAUGES__Get_Omega,
                            GAUGES__Get_wind_speed, GAUGES__Get_wind_dir);
        END;
{}
{}
PROCEDURE TRANSMITTER__Broadcast_detail;
        BEGIN
           WHILE NOT (MEMORY__Is_memory_empty) DO
              BEGIN
                 WRITELN (Transmitter, MEMORY__Get_historic_reading);
              END;
        END;
{}
{}
{}
{*****************************************************}
{}
{}
{*** OBJECT Receiver *****************************}
{}
{}
      TYPE
              Request_type = (None, Sos, Sosoth, Air, Ship);
{}
      VAR [STATIC]
              Current_request : Request_type;
              Receiver : I_O_Type;
{}
{}
PROCEDURE RECEIVER__Start_receiver;
{}
         BEGIN
            RESET (Receiver);
         END;
{}
{}
PROCEDURE RECEIVER__Receive_next_request;
{}
         BEGIN
            READLN (Receiver, Current_request);
         END;
{}
{}
FUNCTION RECEIVER__What_is_curr_request : Request_type;
{}
         BEGIN
            RECEIVER__What_is_curr_request := Current_request;
         END;
{}
{}
{}
{*****************************************************
{}
{}
{}
```

```
{ }
{******* Clock Object ***********************************}
{ }
{ }
        VAR [STATIC] Seconds : INTEGER;
{ }
        Procedure CLOCK__Start_clock;
{ }
                BEGIN
                   Seconds := 1;
                END;
{ }
{ }
         PROCEDURE CLOCK__Increment_clock;
{ }
                 BEGIN
                    Seconds := Seconds + 1;
                 END;
{ }
{ }
         FUNCTION CLOCK__Send_time : INTEGER;
{ }
                  BEGIN
                     CLOCK__Send_time := Seconds;
                  END;
{ }
{ }
{ }
{ }
{*** AAS Main Process ****************************}
{ }
{ }
{ }
        VAR [STATIC] set_sos : BOOLEAN;
                             F : TEXT;   { Do not alter this line }
{ }
{ }
BEGIN
   MEMORY__Init_memory;
   CLOCK__Start_clock;
   GAUGES__Start_sensors;
   TRANSMITTER__Start_transmitter;
   RECEIVER__Start_receiver;
   Set_sos := FALSE;
{ }
   FOR VAR X := 1 TO 169 DO BEGIN  { ** DO NOT ALTER THIS LINE ** }
{ }
      CLOCK__Increment_clock;
      RECEIVER__Receive_next_request;
      IF (RECEIVER__what_is_curr_request) = None Then
         BEGIN
            IF ((CLOCK__Send_time) MOD 10 = 0) THEN
               BEGIN
                  IF Set_sos THEN
                     TRANSMITTER__broadcast_sos;
                  MEMORY__Store_reading (GAUGES__Get_temp_1);
                  MEMORY__Store_reading (GAUGES__Get_temp_2);
                  MEMORY__Store_reading (GAUGES__Get_[illegible]);
               END;
            IF ((CLOCK__Send_time) MOD [illegible] = 0) THEN
               BEGIN
                  MEMORY__Store_reading (GAUGES__[illegible]_wind_speed);
                  MEMORY__Store_reading (GAUGES__Get_wind_[illegible]);
               END;
            IF ((CLOCK__Send_time) MOD [illegible] = 0) THEN
               TRANSMITTER__Broadcast_[illegible];
```

```
        END
      ELSE
        CASE (RECEIVER__What_is_curr_request) OF
             Sos : Set_sos := TRUE;
             Air : TRANSMITTER__Broadcast_detail;
            Ship : TRANSMITTER__Broadcast_detail;
          Sosoff : Set_sos := FALSE;
        END; {case}
{}
   END;   { ** DO NOT ALTER THIS LINE ** }
   { ** DO NOT ALTER THIS LINE ** }ASSIGN(F,'RUN.OK');REWRITE(F);CLOSE(F);
END
{**************************************************}.
```

Military Address Problem
(Functional Decomposition)

```
Program MADDS (Data_file, Printer, Input); {{DEBUG+}{linesize:1[illegible]
{}
{}
{}
{}
      CONST
           blank = ' ';
{}
      TYPE
{}
         String_4_type = LSTRING (4);
         String_10_type = LSTRING (10);
         String_15_type = LSTRING (15);
         String_20_type = LSTRING (20);
         Grade_type = (Private, Corporal, Sargeant, Lieutenant,
                       Captain, Major, Colonel, General,
                       Unknown, None, All);
         Zip_type = String_10_type;
{}
         File_structure = RECORD
                                    Title : String_4_type;
                                Last_name : String_15_type;
                               Given_name : String_20_type;
                                   Branch : String_20_type;
                                  Command : String_20_type;
                                   Street : String_20_type;
                                     City : String_20_type;
                                    State : String_20_type;
                                  Country : String_15_type;
                                      Zip : String_10_type;
                                    Grade : Grade_type;
                          END;
{}
      VAR
           F : TEXT; { DO NOT ALTER THIS LINE }
           Low_zip, High_zip, Zip_state : Zip_type;
           Low_grade, High_grade, Grade_state : Grade_type;
           Curr_record : File_structure;
           EOFile : BOOLEAN;
           Data_file : TEXT;
           Printer : TEXT;
           Index : INTEGER;
           Pvt_count, Corp_count, Sgt_count,
           Lt_count, Capt_count, Major_count,
           Col_count, Gen_count                   : INTEGER;
           Zip_count : INTEGER;
{}
{
{
{
{}
  [illegible] criteria (VAR Low_z,
                                         High_z : Zip_type;
                                     VAR Low_g,
                                         High_g : Grade_type);
{
{}
           VAR
              In_string : String_20_type;
{}
{
```

```
{}
{}
               PROCEDURE Convert_Instring_to_grade_type
                                (Instrng : String_20_type;
                                 VAR  G : Grade_type);
{}
                     BEGIN
                        IF Instrng = 'Private' THEN
                           G := Private
                        ELSE IF Instrng = 'Corporal' THEN
                            G := Corporal
                        ELSE IF Instrng = 'Lieutenant' THEN
                            G := Lieutenant
                        ELSE IF Instrng = 'Sargeant' THEN
                            G := Sargeant
                        ELSE IF Instrng = 'Captain' THEN
                            G := Captain
                        ELSE IF Instrng = 'Major' THEN
                            G := Major
                        ELSE IF Instrng = 'Colonel' THEN
                            G := Colonel
                        ELSE IF Instrng = 'General' THEN
                            G := General
                        ELSE
                            G := Unknown;
                     END;
{}
{}
{}
{}
               FUNCTION Valid_zip (Z : Zip_type) : BOOLEAN;
{}
                   BEGIN
                    Valid_zip := TRUE;
                    FOR Inde  := 1 TO ORD (Z.LEN) DO
                      IF NOT (Z [Inde ] IN [ '0' .. '9', Blank])
                         THEN Valid_zip := FALSE;
                   END;
{}
{}
               FUNCTION Valid_grade (G : Grade_type) : BOOLEAN;
{}
                   BEGIN
                     Valid_grade := TRUE;
                     IF NOT (G IN [Private..General])
                        THEN Valid_grade := FALSE;
                   END;
{}
{}
             BEGIN
               REPEAT
                  Low_z := NULL;
                  WRITE (Printer, 'Enter low postal code, ');
                  WRITE (Printer, 'or just RETURN for nil: ');
                  READLN (INPUT, Low_z);
                  WRITELN (Printer);
               UNTIL Valid_zip (Low_z);
               IF NOT (Low_z = NULL) THEN
                  REPEAT
                     High_z := NULL;
                     WRITE (Printer, 'Enter high postal code, ');
                     WRITE (Printer, 'or just RETURN for single postal code: ');
:
                     READLN (INPUT, High_z);
                     WRITELN (Printer);
                  UNTIL Valid_zip (High_z);
```

```
            IF Low_z = NULL THEN
               BEGIN
                  Low_z  :=            ';
                  High_z := '9999999999';
               END
             ELSE IF High_z = NULL THEN
                     High_z := Low_z;
           REPEAT
              Low_g := None;
              High_g := None;
              WRITE (Printer, 'Enter low O-Grade, ');
              WRITE (Printer, 'or just RETURN for ALL: ');
              READLN (INPUT, In_string);
              WRITELN (Printer);
              IF In_string = NULL THEN
                 BEGIN
                    Low_g := Private;
                    High_g := General;
                 END
               ELSE
                   Convert_instring_to_grade_type (In_string,
                                                    Low_g);
           UNTIL Valid_grade (Low_g);
           IF NOT (High_g = General) THEN
              REPEAT
                 WRITE (Printer, 'Enter high O-Grade, ');
                 WRITE (Printer, 'or just RETURN for single O-Grade: ');
                 READLN (INPUT, In_string);
                 WRITELN (Printer);
                 IF In_string = NULL THEN
                    High_g := Low_g
                  ELSE
                    Convert_instring_to_grade_type (In_string,
                                                     High_g);
              UNTIL Valid_grade (High_g);
         END;
{}
{}
{}
{}
{}

     PROCEDURE Initialize_counters;
{}
          BEGIN
             Pvt_count := 0;
             Corp_count := 0;
             Sgt_count := 0;
             Lt_count := 0;
             Capt_count := 0;
             Major_count := 0;
             Col_count := 0;
             Gen_count := 0;
          END;
{}
{}
{}

{}
     PROCEDURE Read_record (VAR Curr_rec : File_structure;
                            VAR End_of_file : Boolean);
{}
         BEGIN
           End_of_file := FALSE;
           WITH Curr_rec DO
            BEGIN
               READLN (Data_file, Title);
```

```
                  IF Title <> '****' THEN
                     BEGIN
                        READLN (Data_file, Last_name);
                        READLN (Data_file, Given_name);
                        READLN (Data_file, Branch);
                        READLN (Data_file, Command);
                        READLN (Data_file, Street);
                        READLN (Data_file, City);
                        READLN (Data_file, State);
                        READLN (Data_file, Country);
                        READLN (Data_file, Zip);
                        READLN (Data_file, Grade);
                     END
                  ELSE
                     End_of_file := TRUE;
               END;
            END;
{}
{}
{}
{}
      FUNCTION Matches (Low_zip, High_zip : Zip_type;
                        Low_grade, High_grade : Grade_type;
                        Curr_Rec : File_structure)             : BOOLEAN;
{}
{}
            BEGIN
               Matches := FALSE;
               IF (Curr_Rec.Zip >= Low_zip) AND
                  (Curr_rec.Zip <= High_zip) AND
                  (Curr_rec.Grade >= Low_grade) AND
                  (Curr_rec.Grade <= High_grade)
                     THEN  Matches := TRUE;
            END;
{}
{}
{}
   PROCEDURE Process_match;
{}
{}
         PROCEDURE Increment_grade_counters (Counter : Grade_type);

               BEGIN
                  Case Counter of
                      Private : Pvt_count := Pvt_count + 1;
                     Corporal : Corp_count := Corp_count + 1;
                     Sargeant : Sgt_count := Sgt_count + 1;
                   Lieutenant : Lt_count := Lt_count + 1;
                      Captain : Capt_count := Capt_count + 1;
                        Major : Major_count := Major_count + 1;
                      Colonel : Col_count := Col_count + 1;
                      General : Gen_count := Gen_count + 1;
                  End;
               END;
{}
{}
{}
         PROCEDURE Print_label;
{}
{}
               PROCEDURE Write_given_name [illegible]

                        VAR Index , Index_2 : INTEGER;

                  BEGIN
                     Index := 1;
```

```
                  WHILE G_name [Index] <> Blank DO
                    BEGIN
                       WRITE (Printer, G_name [Index]);
                       Index := Index + 1;
                    END;
                  Index := Index + 1;
                  IF G_name [Index] <> '*' then
                    BEGIN
                      WRITE (Printer, Blank);
                      FOR Index2 := Index TO ORD (G_name.LEN) DO
                          WRITE (Printer, G_name [Index2]);
                    END;
                END;
{}
{}
         BEGIN
            WITH Curr_record DO
              BEGIN
                 WRITE (Printer, Title, Blank);
                 Write_given_name (Given_name);
                 WRITE (Printer, Blank, Last_name);
                 WRITELN (Printer);
                 WRITELN (Printer, Branch);
                 WRITELN (Printer, Command);
                 WRITELN (Printer, City, ',', Blank, State);
                 WRITELN (Printer, Country, Blank, Zip);
                 WRITELN (Printer);
                 WRITELN (Printer);
                 WRITELN (Printer);
                 WRITELN (Printer);
                 WRITELN (Printer);
              END;
         END;
{}
{}
     BEGIN
         IF NOT (Curr_record.zip = Zip_state) THEN
           BEGIN
              WRITELN (Printer);
              WRITELN (Printer, 'Total for zip ', Zip_state,
                                          ': ', Zip_count);
              WRITELN (Printer);
              WRITELN (Printer);
              WRITELN (Printer);
              Zip_state := Curr_record.zip;
              Zip_count := 0;
           END;
         Zip_count := Zip_count + 1;
         Increment_Grade_counters (Curr_record.grade);
         Print_label;
     END;
{}
{}
{}
{}
{}
{}
      PROCEDURE Print_grade_totals (Low_gr, High_gr : Grade_type);

              VAR
                  This_Grade : Grade_type;
{}
{}
              PROCEDURE Print_this_total
                                     (Grade_[illegible] : [illegible];
                                      Total : INTEGER);
```

```
                              BEGIN
                                 WRITELN (Printer, 'Total for ',
                                          Grade_string, ' is: ', Total);
                              END;
{}
                   BEGIN
                      FOR This_grade := Low_gr TO High_gr DO
                         IF This_grade = Private THEN
                            Print_this_total ('Private   ', Pvt_count)
                         ELSE IF This_grade = Corporal THEN
                            Print_this_total ('Corporal  ', Corp_count)
                         ELSE IF This_grade = Sargeant THEN
                            Print_this_total ('Sargeant  ', Sgt_count)
                         ELSE IF This_grade = Lieutenant THEN
                            Print_this_total ('Lieutenant', Lt_count)
                         ELSE IF This_grade = Captain THEN
                            Print_this_total ('Captain   ', Capt_count)
                         ELSE IF This_grade = Major THEN
                            Print_this_total ('Major     ', Major_count)
                         ELSE IF This_grade = Colonel THEN
                            Print_this_total ('Colonel   ', Col_count)
                         ELSE IF This_grade = General THEN
                            Print_this_total ('General   ', Gen_count);
                   END;
{}
{}
{}
BEGIN
   RESET (Data_file);
   RESET (INPUT);
   REWRITE (Printer);
   Initialize_counters;
   Zip_count := 0;
   EOFile := TRUE;
   Select_Criteria (Low_zip, High_zip, Low_grade, High_grade);
   Read_Record (Curr_record, EOFile);
   Zip_state := Curr_record.Zip;
   WHILE NOT EOFile DO
     BEGIN
        IF Matches (Low_zip, High_zip,
                    Low_grade, High_grade, Curr_record) THEN
           Process_match;
        Read_Record (Curr_record, EOFile);
     END; { While }
   WRITELN (Printer);
   WRITELN (Printer, 'Total for zip ', Zip_state, ': ', Zip_count);
   WRITELN (Printer);
   WRITELN (Printer);
   WRITELN (Printer);
   Print_grade_totals (Low_grade, High_grade);
   CLOSE (Data_file);
   CLOSE (Printer);
   CLOSE (INPUT);
   { ***  Do NOT alter this line  *** [illegible] :REWRITE [illegible]
*}
END.
```

Military Address Problem
(In-Line)

```
Program MADBS (Data_file, Printer, Input);  {$DEBUG+}{$linesize:132}
{}
{}
{}
{}
      CONST
            Blank =   ' ';
{}
      TYPE
{}
          String_4_type = LSTRING (4);
          String_10_type = LSTRING (10);
          String_15_type = LSTRING (15);
          String_20_type = LSTRING (20);
          Grade_type = (Private, Corporal, Sargeant, Lieutenant,
                        Captain, Major, Colonel, General,
                        Unknown, None, All);
          Zip_type = String_10_type;
{}
          File_structure = RECORD
                                   Title : String_4_type;
                               Last_name : String_15_type;
                              Given_name : String_20_type;
                                  Branch : String_20_type;
                                 Command : String_20_type;
                                  Street : String_20_type;
                                    City : String_20_type;
                                   State : String_20_type;
                                 Country : String_15_type;
                                     Zip : String_10_type;
                                   Grade : Grade_type;
                           END;
{}
      VAR
            F : TEXT; { Do NOT alter this line }
            Low_zip, High_zip, Zip_state : Zip_type;
            Low_grade, High_grade, Grade_state, This_grade : Grade_type;
            Curr_record : File_structure;
            EOFile : BOOLEAN;
            Data_file : TEXT;
            Printer : TEXT;
            Index : INTEGER;
            Index2 : INTEGER;
            Pvt_count, Corp_count, Sgt_count,
            Lt_count, Capt_count, Major_count,
            Col_count, Gen_count                  : INTEGER;
            Zip_count : INTEGER;
            In_string : String_20_type;
            Valid_zip, Valid_grade  :  BOOLEAN;
{}
{}
{}
{}
{}
{}
BEGIN
   RESET (data_file);
   RESET (INPUT);
   REWRITE (Printer);
   Pvt_count := 0;
   Corp_count := 0;
```

```
Sgt_count := 0;
Lt_count := 0;
Capt_count := 0;
Major_count := 0;
Col_count := 0;
Gen_count := 0;
Zip_count := 0;
EOFile := TRUE;
REPEAT
   Low_zip := NULL;
   WRITE (Printer, 'Enter low postal code, ');
   WRITE (Printer, 'or just RETURN for All: ');
   READLN (INPUT, Low_zip);
   WRITELN (Printer);
   Valid_zip := TRUE;
   FOR Index := 1 TO ORD (Low_zip.LEN) DO
        IF NOT (Low_zip [Index] IN ['0' .. '9', Blank])
           THEN Valid_zip := FALSE;
UNTIL Valid_zip;
IF NOT (Low_zip = NULL) THEN
REPEAT
   High_zip := NULL;
   WRITE (Printer, 'Enter high postal code, ');
   WRITE (Printer, 'or just RETURN for single postal code: ');
   READLN (INPUT, High_zip);
   WRITELN (Printer);
   Valid_zip := TRUE;
   FOR Index := 1 TO ORD (High_zip.LEN) DO
        IF NOT (High_zip [Index] IN ['0' .. '9', Blank])
           THEN Valid_zip := FALSE;
UNTIL Valid_zip;
IF Low_zip = NULL THEN
   BEGIN
      Low_zip  := '          ';
      High_zip := '99999-9999';
   END
 ELSE IF High_zip = NULL THEN
        High_zip := Low_zip;
        REPEAT
           Low_grade := None;
           High_grade := None;
           WRITE (Printer, 'Enter low O-Grade, ');
           WRITE (Printer, 'or just RETURN for ALL: ');
           READLN (INPUT, In_string);
           WRITELN (Printer);
           IF In_string = NULL THEN
              BEGIN
                 Low_grade := Private;
                 High_grade := General;
              END
             ELSE
              BEGIN
                 IF In_string = 'Private' THEN
                    Low_grade := Private
                 ELSE IF In_string = 'Corporal' THEN
                      Low_grade := Corporal
                 ELSE IF In_string = 'Lieutenant' THEN
                      Low_grade := Lieutenant
                 ELSE IF In_string = 'Sargeant' THEN
                      Low_grade := Sargeant
                 ELSE IF In_string = 'Captain' THEN
                      Low_grade := Captain
                 ELSE IF In_string = 'Major' THEN
                      Low_grade := Major
                 ELSE IF In_string = 'Colonel' THEN
                      Low_grade := Colonel
```

```
                ELSE IF In_string = 'General' THEN
                   Low_grade := General
                ELSE
                   Low_grade := Unknown;
             END;
          Valid_grade := TRUE;
          IF NOT (Low_grade IN [Private..General])
             THEN Valid_grade := FALSE;
       UNTIL Valid_grade;
       IF NOT (High_grade = General) THEN
          REPEAT
             WRITE (Printer, 'Enter high O-Grade, ');
             WRITE (Printer, 'or just RETURN for single O-Grade: ');
             READLN (INPUT, In_string);
             WRITELN (Printer);
             IF In_string = NULL THEN
                High_grade := Low_grade
              ELSE
                BEGIN
                   IF In_string = 'Private' THEN
                      High_grade := Private
                   ELSE IF In_string = 'Corporal' THEN
                      High_grade := Corporal
                   ELSE IF In_string = 'Lieutenant' THEN
                      High_grade := Lieutenant
                   ELSE IF In_string = 'Sergeant' THEN
                      High_grade := Sergeant
                   ELSE IF In_string = 'Captain' THEN
                      High_grade := Captain
                   ELSE IF In_string = 'Major' THEN
                      High_grade := Major
                   ELSE IF In_string = 'Colonel' THEN
                      High_grade := Colonel
                   ELSE IF In_string = 'General' THEN
                      High_grade := General
                   ELSE
                      High_grade := Unknown;
                END;
             Valid_grade := TRUE;
             IF NOT (High_grade IN [Private..General])
                THEN Valid_grade := FALSE;
          UNTIL Valid_grade;
EOFile := FALSE;
WITH Curr_record DO
   BEGIN
      READLN (Data_file, Title);
      IF Title <> '****' THEN
         BEGIN
            READLN (Data_file, Last_name);
            READLN (Data_file, Given_name);
            READLN (Data_file, Branch);
            READLN (Data_file, Command);
            READLN (Data_file, Street);
            READLN (Data_file, City);
            READLN (Data_file, State);
            READLN (Data_file, Country);
            READLN (Data_file, Zip);
            READLN (Data_file, Grade);
         END
       ELSE
         EOFile := TRUE;
   END;
Zip_state := Curr_record.Zip;
WHILE NOT EOFile DO
  BEGIN
     IF (Curr_record.Zip ... Low_zip) AND
```

```
(Curr_record.zip <= High_zip) AND
(Curr_record.Grade >= Low_grade) AND
(Curr_record.Grade <= High_grade) THEN
  BEGIN
    IF NOT (Curr_record.zip = Zip_state) THEN
      BEGIN
        WRITELN (Printer);
        WRITELN (Printer, 'Total for zip ', Zip_state,
                          ': ', Zip_count);
        WRITELN (Printer);
        WRITELN (Printer);
        WRITELN (Printer);
        Zip_state := Curr_record.zip;
        Zip_count := 0;
      END;
    Zip_count := Zip_count + 1;
    Case Curr_record.Grade of
         Private : Pvt_count := Pvt_count + 1;
        Corporal : Corp_count := Corp_count + 1;
        Sargeant : Sgt_count := Sgt_count + 1;
      Lieutenant : Lt_count := Lt_count + 1;
         Captain : Capt_count := Capt_count + 1;
           Major : Major_count := Major_count + 1;
         Colonel : Col_count := Col_count + 1;
         General : Gen_count := Gen_count + 1;
      END;
    WITH Curr_record DO
      BEGIN
        WRITE (Printer, Title, Blank);
        index := 1;
        WHILE Given_name [Index] <> Blank do
          BEGIN
            WRITE (Printer, Given_name [Index]);
            Index := Index + 1;
          END;
        Index := Index + 1;
        IF Given_name [Index] <> '*' then
          BEGIN
            WRITE (Printer, Blank);
            FOR Index2 := Index TO ORD (Given_name.LEN) DO
                WRITE (Printer, Given_name [Index2]);
          END;
        WRITE (Printer, Blank, Last_name);
        WRITELN (Printer);
        WRITELN (Printer, Branch);
        WRITELN (Printer, Command);
        WRITELN (Printer, City, ',', Blank, State);
        WRITELN (Printer, Country, Blank, Zip);
        WRITELN (Printer);
        WRITELN (Printer);
        WRITELN (Printer);
        WRITELN (Printer);
        WRITELN (Printer);
      END;
    EOFile := FALSE;
    WITH Curr_record DO
      BEGIN
        READLN (Data_file, Title);
        IF Title <> '****' THEN
          BEGIN
            READLN (Data_file, Last_name);
            READLN (Data_file, Given_name);
            READLN (Data_file, Branch);
            READLN (Data_file, Command);
            READLN (Data_file, Street);
            READLN (Data_file, City);
```

```
                                    READLN (Data_file, State);
                                    READLN (Data_file, Country);
                                    READLN (Data_file, Zip);
                                    READLN (Data_file, Grade);
                                 END
                              ELSE
                                 EOFile := TRUE;
                           END;
                     END;
            END;
      WRITELN (Printer);
      WRITELN (Printer,  Total for zip  , Zip_state,  :  , Zip_count);
      WRITELN (Printer);
      WRITELN (Printer);
      WRITELN (Printer);
      FOR This_grade := Low_grade TO High_grade DO
         IF This_grade = Private THEN
            WRITELN (Printer,  Total for Private      is  ,
                     Pvt_count)
         ELSE IF This_grade = Corporal THEN
            WRITELN (Printer, 'Total for Corporal     is  ,
                     Corp_count)
         ELSE IF This_grade = Sargeant THEN
            WRITELN (Printer,  Total for Sargeant     is  ,
                     Sgt_count)
         ELSE IF This_grade = Lieutenant THEN
            WRITELN (Printer,  Total for Lieutenant is  ,
                     Lt_count)
         ELSE IF This_grade = Captain THEN
            WRITELN (Printer,  Total for Captain      is  ,
                     Capt_count)
         ELSE IF This_grade = Major THEN
            WRITELN (Printer,  Total for Major        is  ,
                     Major_count)
         ELSE IF This_grade = Colonel THEN
            WRITELN (Printer,  Total for Colonel      is  ,
                     Col_count)
         ELSE IF This_grade = General THEN
            WRITELN (Printer,  Total for General      is  ,
                     Gen_count);
      CLOSE (Data_file);
      CLOSE (Printer);
      CLOSE (INPUT);
      { ***  DO NOT ALTER THIS LINE  *** }ASSIGN(F, RUN.OK );REWRITE(F);CLOSE(F);
   END.
```

Military Address Problem
(Object-Oriented)

```
PROGRAM Mauds (Data_file, Printer, Input);    {$debug+} {$lines=[illegible]
{}
{}
{}
    CONST
          Blank = ' ';
{}
    TYPE
          Grade_type = (Private, Corporal, Sargeant, Lieutenant,
                        Captain, Major, Colonel, General,
                        Unknown, None, All);
          Zip_type = LSTRING (20);
{}
{}
{}
{}
{    *****************************************************}
{     OBJECT: Printer object ****************************}
{}
{}
    VAR
          Printer : TEXT;
{}
{}
     PROCEDURE PRINTER__open_printer;
          BEGIN
             REWRITE (Printer);
          END;
{}
{}
{}
{    *****************************************************}
{     OBJECT: User input OBJECT    **********************}
{}
{}
        VAR [STATIC]
                    Low_zip, High_zip : Zip_type;
                    Low_grade, High_grade : Grade_type;
{}
{}
{}
    PROCEDURE USER__Select_Criteria;
{}
{}
                 TYPE
                    String_20_type = LSTRING (20);
{}
                 VAR
                    In_string : String_20_type;
{}
{}
                 PROCEDURE Convert_Instring_to_Grade_type
                                  (Instrng : String_20_type;
                                   VAR  G : Grade_type);
{}
                         BEGIN
                            IF Instrng = 'Private' THEN
                               G := Private
                            ELSE IF Instrng = 'Corporal' THEN
                               G := Corporal
                            ELSE IF Instrng = 'Lieutenant' THEN
```

```
                                G := Lieutenant
                           ELSE IF Instrng = 'Sargeant' THEN
                                G := Sargeant
                           ELSE IF Instrng = 'Captain' THEN
                                G := Captain
                           ELSE IF Instrng = 'Major' THEN
                                G := Major
                           ELSE IF Instrng = 'Colonel' THEN
                                G := Colonel
                           ELSE IF Instrng = 'General' THEN
                                G := General
                           ELSE
                                G := Unknown;
                        END;



          FUNCTION Valid_zip (Z : Zip_type) : BOOLEAN;

                BEGIN
                 Valid_zip := TRUE;
                 FOR VAR Index := 1 TO ORD (Z.LEN) DO
                   IF NOT (Z [Index] IN ['0' .. '9', Blank])
                      THEN Valid_zip := FALSE;
                END;

          FUNCTION Valid_grade (G : Grade_type) : BOOLEAN;

                BEGIN
                  Valid_grade := TRUE;
                  IF NOT (G IN [Private..General])
                     THEN Valid_grade := FALSE;
                END;


    BEGIN
      REPEAT
         Low_zip := NULL;
         WRITE (Printer, 'Enter low postal code, ');
         WRITE (Printer, 'or just RETURN for all: ');
         READLN (INPUT, Low_zip);
         WRITELN (Printer);
      UNTIL Valid_zip (Low_zip);
      IF NOT (Low_zip = NULL) THEN
         REPEAT
            High_zip := NULL;
            WRITE (Printer, 'Enter high postal code, ');
            WRITE (Printer, 'or just RETURN for single postal code: ');
            READLN (INPUT, High_zip);
            WRITELN (Printer);
         UNTIL Valid_zip (High_zip);
      IF Low_zip = NULL THEN
         BEGIN
            Low_zip  := '         ';
            High_zip := '999999999';
         END
       ELSE IF High_zip = NULL THEN
              High_zip := Low_zip;
      REPEAT
         Low_grade  := None;
         High_grade := None;
         WRITE (Printer, 'Enter low Grade, ');
         WRITE (Printer, 'or just RETURN for ALL: ');
         READLN (INPUT, In_string);
         WRITELN (Printer);
```

```
            IF In_string = NULL THEN
               BEGIN
                  Low_grade := Private;
                  High_grade := General;
               END
             ELSE
                 Convert_instring_to_grade_type (In_string,
                                                  Low_grade);
         UNTIL Valid_grade (Low_grade);
         IF NOT (High_grade = General) THEN
            REPEAT
               WRITE (Printer, 'Enter high O-Grade, ');
               WRITE (Printer, 'or just RETURN for single O-Grade: ');
               READLN (INPUT, In_string);
               WRITELN (Printer);
               IF In_string = NULL THEN
                  High_grade := Low_grade
                ELSE
                  Convert_instring_to_grade_type (In_string,
                                                   High_grade);
            UNTIL Valid_grade (High_grade);
      END;
{}
{}
{}
{   ***********************************************************}
{}
{   OBJECT:   File object   **********************************}
{}
{}
    TYPE
{}
      String_4_type = LSTRING (4);
      String_10_type = LSTRING (10);
      String_15_type = LSTRING (15);
      String_20_type = LSTRING (20);
{}
{}
{}
      File_structure = RECORD
                                  Title : String_4_type;
                              Last_name : String_15_type;
                             Given_name : String_20_type;
                                 Branch : String_20_type;
                                Command : String_20_type;
                                 Street : String_20_type;
                                   City : String_20_type;
                                  State : String_20_type;
                                Country : String_15_type;
                                    Zip : Zip_type;
                                  Grade : Grade_type;
                       END;
{}
{}
    VAR
      Data_file : TEXT;
      Curr_record : File_structure;
{}
    PROCEDURE FILE_open_file;
{}
         BEGIN
            RESET (Data_file);
         END;
{}
```

```
{}
     FUNCTION FILE__Find_match : BOOLEAN;
{}
                    VAR EOFile : BOOLEAN;
{}
{}
                 FUNCTION Matches : BOOLEAN;
{}
                        BEGIN
                           Matches := FALSE;
                           IF (Curr_Record.Zip >= Low_zip) AND
                              (Curr_record.Zip <= High_zip) AND
                              (Curr_record.Grade >= Low_grade) AND
                              (Curr_record.Grade <= High_grade)
                                  THEN  Matches := TRUE;
                        END;
{}
{}
{}
      BEGIN
        EOFile := FALSE;
            WITH Curr_record DO
             BEGIN
                REPEAT
                   READLN (Data_file, Title);
                   IF Title <> '****' THEN
                      BEGIN
                         READLN (Data_file, Last_name);
                         READLN (Data_file, Given_name);
                         READLN (Data_file, Branch);
                         READLN (Data_file, Command);
                         READLN (Data_file, Street);
                         READLN (Data_file, City);
                         READLN (Data_file, State);
                         READLN (Data_file, Country);
                         READLN (Data_file, Zip);
                         READLN (Data_file, Grade);
                      END
                   ELSE
                      EOFile := TRUE;
                UNTIL Matches OR EOFile;
                IF Matches AND (NOT EOFile)
                   THEN FILE__Find_match := TRUE
                   ELSE FILE__Find_match := FALSE;
            END;
      END;
{}
{}
{}
{}
{}
     FUNCTION FILE__Send_Title : String_4_type;
{}
           BEGIN
              FILE__Send_Title := Curr_record.Title;
           END;
{}
{}
     FUNCTION FILE__Send_last_name : String_1_ type;
{}
           BEGIN
              FILE__Send_last_name := Curr_record.Last_name;
           END;
{}
{}
     FUNCTION FILE__Send_given_name : String_  type;
```

```
{ }
          BEGIN
             FILE__Send_given_name := Curr_record.given_name;
          END;
{ }
{ }
     FUNCTION FILE__Send_Branch : String_20_type;
          BEGIN
             FILE__Send_Branch := Curr_record.Branch;
          END;
{ }
{ }
     FUNCTION FILE__Send_Command : String_20_type;
{ }
          BEGIN
             FILE__Send_command := Curr_record.Command;
          END;
{ }
{ }
     FUNCTION FILE__Send_City : String_20_type;
{ }
          BEGIN
             FILE__Send_city := Curr_record.City;
          END;
{ }
{ }
     FUNCTION FILE__Send_State : String_20_type;
{ }
          BEGIN
             FILE__Send_state := Curr_record.State;
          END;
{ }
{ }
     FUNCTION FILE__Send_Country : String_15_type;
{ }
          BEGIN
             FILE__Send_country := Curr_record.Country;
          END;
{ }
{ }
     FUNCTION FILE__Send_zip : Zip_type;
{ }
          BEGIN
             FILE__Send_zip := Curr_record.Zip;
          END;
{ }
{ }
     FUNCTION FILE__Send_grade : Grade_type;
{ }
           BEGIN
             FILE__Send_grade := Curr_record.Grade;
           END;
{ }
{ }
     PROCEDURE FILE__Close_files;
{ }
           BEGIN
              CLOSE (Data_file);
           END;
{ }
{    ******************************************************** }
{ }
{ }
{ }
{ }
```

```
{ }
{    ***************************************************************
{ }
{    OBJECT:    Label object      ****************************** }
{ }
{ }
{ }
{ }

     PROCEDURE LABEL__Print_Label;

{ }
{ }
                    FUNCTION Convert_given_name (Item : String_20_type)
                                                   : String_20_type;
{ }
                         VAR
                              Temp : String_20_type;
                              Index : Integer;
{ }
                         BEGIN
                            Temp := NULL;
                            Index := 1;
                            WHILE Item [Index]      Blank DO
                              BEGIN
                                 CONCAT (Temp, Item [Index]);
                                 Index := Index + 1;
                              END;
                            Index := Index + 1;
                            IF (Item [Index]       '*' ) THEN
                              BEGIN
                                 CONCAT (Temp, Blank);
                                 REPEAT
                                    CONCAT (Temp, Item [Index]);
                                    Index := Index + 1;
                                 UNTIL CHR (Index)     Item [0];
                              END; { if }
                            Convert_given_name := Temp;
                         END;
{ }
{ }
{ }
       BEGIN
          WRITELN (Printer, FILE__Send_title, Blank,
                        Convert_given_name (FILE__Send_Given_name),
                        Blank, FILE__Send_last_name);
          WRITELN (Printer, FILE__Send_branch);
          WRITELN (Printer, FILE__Send_command);
          WRITELN (Printer, FILE__Send_city, ', ', FILE__Send_state);
          WRITELN (Printer, FILE__Send_country, Blank, FILE__Send_zip);
          WRITELN (Printer);
          WRITELN (Printer);
          WRITELN (Printer);
          WRITELN (Printer);
          WRITELN (Printer);
       END;
{ }
{ }
{ }
{    ***************************************************************}
{    OBJECT: Counter object   ******************************}
{ }
{ }
{ }
         VAR [STATIC]
               Pvt_count, Corp_count, [illegible],
               Lt_count, Capt_count, Major_count,
                Col_count, Gen_count, [illegible] : INTEGER;
```

```
            Zip_state : zip_type;
{}
{}
{}
    PROCEDURE COUNTER__Initialize_counters;
{}
        BEGIN
          Pvt_count := 0;
          Corp_count := 0;
          Sgt_count := 0;
          Lt_count := 0;
          Capt_count := 0;
          Major_count := 0;
          Col_count := 0;
          Gen_count := 0;
          Zip_count := 0;
        END;
{}
{}
{}
    PROCEDURE COUNTER__Set_initial_zip_state;
{}
        BEGIN
          Zip_state := FILE__Send_zip;
        END;
{}
{}
{}
  PROCEDURE COUNTER__Increment_counters;
{}
{}
      PROCEDURE Increment_grade_counters (Counter : Grade_type);

          BEGIN
            Case Counter of
                 Private : Pvt_count := Pvt_count + 1;
                Corporal : Corp_count := Corp_count + 1;
                Sargeant : Sgt_count := Sgt_count + 1;
              Lieutenant : Lt_count := Lt_count + 1;
                 Captain : Capt_count := Capt_count + 1;
                   Major : Major_count := Major_count + 1;
                 Colonel : Col_count := Col_count + 1;
                 General : Gen_count := Gen_count + 1;
            END;
          END;
{}
    BEGIN
      IF NOT ((FILE__Send_zip) = Zip_state) THEN
        BEGIN
          WRITELN (Printer);
          WRITELN (Printer, 'Total for zip ', Zip_state,
                                    ' : ', Zip_count);
          WRITELN (Printer);
          WRITELN (Printer);
          WRITELN (Printer);
          Zip_state := FILE__Send_zip;
          Zip_count := 0;
        END;
      Zip_count := Zip_count + 1;
      Increment_grade_counters (FILE__Send_grade);
    END;
{}
{}
{}
    PROCEDURE COUNTER__Print_grade_totals;
{}
```

```
                 VAR
                     This_grade : grade_type;
{}
{}
                 PROCEDURE Print_this_total
                               (Grade_string : String_20_type;
                                Total : INTEGER);
                        BEGIN
                           WRITELN (Printer, 'Total for ',
                                    Grade_string, ' is: ', Total);
                        END;
{}
            BEGIN
               WRITELN (Printer);
               WRITELN (Printer);
               WRITELN (Printer);
               FOR This_grade := Low_grade TO High_grade DO
                  IF This_grade = Private THEN
                     Print_this_total ('Private    ', Pvt_count)
                  ELSE IF This_grade = Corporal THEN
                     Print_this_total ('Corporal   ', Corp_count)
                  ELSE IF This_grade = Sargeant THEN
                     Print_this_total ('Sargeant   ', Sgt_count)
                  ELSE IF This_grade = Lieutenant THEN
                     Print_this_total ('Lieutenant ', Lt_count)
                  ELSE IF This_grade = Captain THEN
                     Print_this_total ('Captain    ', Capt_count)
                  ELSE IF This_grade = Major THEN
                     Print_this_total ('Major      ', Major_count)
                  ELSE IF This_grade = Colonel THEN
                     Print_this_total ('Colonel    ', Col_count)
                  ELSE IF This_grade = General THEN
                     Print_this_total ('General    ', Gen_count);
            END;
{}
{  *************************************************************}
{}
{}
{}
{     PROGRAM MADDS     }
{}
          VAR [STATIC] Continue : BOOLEAN;
                   F : text; { Do NOT alter this line }
{}
{}
BEGIN
   FILE__Open_files;
   PRINTER__Open_printer;
   COUNTER__Initialize_counters;
   USER__Select_criteria;
   Continue := FILE__Find_match;
   COUNTER__Set_initial_zip_state;
   WHILE Continue DO
     BEGIN
        COUNTER__Increment_counters;
        LABEL__Print_label;
        Continue := FILE__Find_match;
     END;
   WRITELN (Printer);
   WRITELN (Printer, 'Total for zip ', Zip_state, ' : ', Zip_count);
   COUNTER__Print_grade_totals;
   FILE__Close_files;
   { ***  Do NOT alter This Line  ***ASSIGN(F, [illegible]);REWRITE(F);[illegible] }
END.
```

Student Transactions Problem
(Functional Decomposition)

```
PROGRAM Classlist (Permfile, Transfile, Printer); {$debug+} {$linesize:132}
{}
{}
{}
{}
    TYPE
        Name_array = PACKED ARRAY [1..35] OF CHAR;
        SS_array = PACKED ARRAY [1..10] OF CHAR;
        Link =  Object;
{}
        Object = RECORD
                     Next : Link;
                     Student_name : Name_array;
                     Social_security : SS_array;
                 END;
{}
{}
    VAR
        F : TEXT; { Do NOT alter this line }
        Permfile : TEXT;
        Transfile : TEXT;
        Printer : TEXT;
        Command : CHAR;
        Name : Name_array;
        SS_number : SS_array;
        Column : INTEGER;
        First : Link;
{}
{}
{}
{}
      PROCEDURE Skip_lines (How_many : INTEGER);
{}
              VAR
                  Index : INTEGER;
{}
             BEGIN
                For index := 1 TO How_many Do
                    WRITELN (Printer);
             END;
{}
{}
{}
{}
      PROCEDURE Read_data_line (VAR A_file : TEXT);
{}
               VAR
                   Ch : CHAR;
{}
              BEGIN
                 READ (A_file, Command);
                 FOR Column := 2 TO 36 DO
                     BEGIN
                        READ (A_file, Ch);
                        Name [Column - 1] := Ch;
                     END;
                 FOR Column := 37 TO 46 DO
                     BEGIN
                        READ (A_file, Ch);
                        SS_number [Column - 36] := Ch;
                     END;
```

```
                READLN (A_file);
              END;
{}
{}
{}
{}
      PROCEDURE Search (VAR Found : BOOLEAN; VAR Q, P : Link);
{}
            BEGIN
              Q := First;
              P := First .Next;
              Found := FALSE;
              WHILE (P <> NIL) AND (NOT Found) DO
                IF (P .Student_name = Name) AND
                   (P^.Social_security = SS_number)
                     THEN Found := TRUE
                     ELSE
                       BEGIN
                         Q := P;
                         P := P .Next;
                       END;
            END;
{}
{}
{}
{}
      PROCEDURE Add_student;
{}
               VAR
                  Q, P : Link;
                  Duplicate : BOOLEAN;
                  X, Y : Link;
{}
               PROCEDURE Insert_after (After_this : Link);
{}
                         VAR
                            Temp : Link;
{}
{}
                        BEGIN
                          NEW (Temp);
                          Temp .Student_name := Name;
                          Temp .Social_security := SS_number;
                          Temp .Next := After_this .Next;
                          After_this .Next := Temp;
                        END;
{}
{}
               FUNCTION Empty_list : BOOLEAN;
{}
                        BEGIN
                          IF First .Next = NIL
                             THEN Empty_list := TRUE
                             ELSE Empty_list := FALSE;
                        END;
{}
{}
          BEGIN
            IF Empty_list THEN
              Insert_after (First)
             ELSE
               BEGIN
                 Search (Duplicate, X, Y);
                 IF Duplicate THEN
                   BEGIN
                     Skip_lines (1);
```

```
                        WRITELN (Printer,
                                 'Duplicate record: Not Added');
                      END
                    ELSE
                      BEGIN
                        Q := First;
                        P := First^.Next;
                        IF Name < P^.Student_name THEN
                          Insert_after (First)
                         ELSE
                          BEGIN
                            WHILE (Name > P^.Student_name) AND
                                  (P^.Next <> NIL) DO
                              BEGIN
                                Q := P;
                                P := Q^.Next;
                              END;
                            IF Name > P^.Student_name
                              THEN Insert_after (P)
                              ELSE Insert_after (Q);
                          END;
                      END;
                  END;
              END;
{}
{}
{}
{}

          PROCEDURE Drop_student;
{}
                VAR
                  Preceeding, Actual : Link;
                  Is_it_there : BOOLEAN;
{}
              BEGIN
                Search (Is_it_there, Preceeding, Actual);
                IF Is_it_there THEN
                  Preceeding^.Next := Actual^.Next
                 ELSE
                  BEGIN
                    Skip_lines (1);
                    WRITELN (Printer,
                            'Student not in class: No drop done.');
                  END;
              END;
{}
{}
{}
{}

          PROCEDURE Inquire;
{}
                 VAR
                    Proceeding, Actual : Link;
                    Is_it_there : BOOLEAN;
{}
              BEGIN
                Search (Is_it_there, Proceeding, Actual);
                IF Is_it_there THEN
                  BEGIN
                    Skip_lines (1);
                    WRITELN (Printer, Name, ' is in the record.');
                  END
                 ELSE
                  BEGIN
                    Skip_lines (1);
                    WRITELN (Printer, Name, ' is NOT in the record.');
```

```
                END;
             END;
{}
{}
{}
{}
      PROCEDURE List;
{}
                VAR
                   Q, P : Link;
{}
         BEGIN
            Q := First;
            P := First .Next;
            Skip_lines (1);
            WHILE P <> NIL DO
               BEGIN
                  WRITELN (Printer, P .Student_name,
                                            P .Social_security);
                  P := P .Next;
               END;
         END;
{}
{}
{}
{}
      PROCEDURE Error;
{}
         BEGIN
            Skip_lines (1);
            WRITELN (Printer,
                         'Invalid command. Line from transaction ',
                         'file ignored.');
         END;
{}
{}
{}
{}
      PROCEDURE Read_in_permanent_file;
{}
         BEGIN
            RESET (Permfile);
            Read_data_line (Permfile);
            WHILE NOT EOF (Permfile) DO
               BEGIN
                  Add_student;
                  Read_data_line (Permfile);
               END;
            CLOSE (Permfile);
         END;
{}
{}
{}
{}
      PROCEDURE Save_permanent_file;
{}
                VAR
                   Q, P : Link;
{}
         BEGIN
            REWRITE (Permfile);
            Q := First;
            P := First .Next;
            WHILE P <> NIL DO
               BEGIN
                  WRITELN (Permfile,     , P .Student_name,
```

```
                                                                 D^.Social_security);
                                   Q := P;
                                   P := P^.Next;
                                END;
                             CLOSE (Permfile);
                          END;
{}
{}
{}
{}
BEGIN
   NEW (First);
   First^.Next := NIL;
   REWRITE (Printer);
   Read_in_permanent_file;
   RESET (Transfile);
   Read_data_line (Transfile);
   WHILE NOT EOF (Transfile) DO
      BEGIN
         CASE Command OF
              'A' : Add_student;
              'D' : Drop_student;
              'I' : Inquire;
              'L' : List;
          END;
         IF NOT (Command IN ['A', 'D', 'I', 'L'])
            THEN Error;
         Read_data_line (Transfile);
      END;
   Save_permanent_file;
   Skip_lines (1);
   WRITELN (Printer, 'Transaction file completed.');
   CLOSE (Transfile);
   CLOSE (Printer);
   { ** Do NOT alter this line ** }ASSIGN(F,'RUN.OF');REWRITE(F);CLOSE(F);
END.
```

Student Transactions Problem
(In-Line)

```
PROGRAM Classlist (Permfile, Transfile, Printer); {$debug+} {$linesize:132}
{}
{}
{}
{}
     TYPE
          Name_array = PACKED ARRAY [1..35] OF CHAR;
          SS_array = PACKED ARRAY [1..10] OF CHAR;
          Link =  Object;
{}
          Object = RECORD
                     Next : Link;
                     Student_name : Name_array;
                     Social_security : SS_array;
                   END;
{}
{}
     VAR
          F : TEXT; { Do NOT alter this line }
          Permfile : TEXT;
          Transfile : TEXT;
          Printer : TEXT;
          Command : CHAR;
          Name : Name_array;
          SS_number : SS_array;
          Column : INTEGER;
          First, Temp, P, Q : Link;
          Found : BOOLEAN;
          Ch : CHAR;
{}
{}
{}
{}
{}
{}
BEGIN
   NEW (First);
   First .Next := NIL;
   REWRITE (Printer);
   RESET (Permfile);
   READ (Permfile, Command);
   FOR Column := 2 TO 36 DO
      BEGIN
         READ (Permfile, Ch);
         Name [Column - 1] := Ch;
      END;
   FOR Column := 37 TO 46 DO
      BEGIN
         READ (Permfile, Ch);
         SS_number [Column - 36] := Ch;
      END;
   READLN (Permfile);
   WHILE NOT EOF (Permfile) DO
      BEGIN
         IF First .Next = NIL THEN
            BEGIN
               NEW (Temp);
               Temp .Student_name := Name;
               Temp .Social_security := SS_number;
               Temp .Next := First .Next;
               First .Next := Temp;
```

```
            END
         ELSE
            BEGIN
               Q := First;
               P := First^.Next;
               Found := FALSE;
               WHILE (P <> NIL) AND (NOT Found) DO
                  IF (P^.Student_name = Name) AND
                     (P^.Social_security = SS_number)
                     THEN Found := TRUE
                     ELSE
                        BEGIN
                           Q := P;
                           P := P^.Next;
                        END;
               IF Found THEN
                  BEGIN
                     WRITELN (Printer);
                     WRITELN (Printer,
                              'Duplicate record: Not Added');
                  END
                ELSE
                  BEGIN
                     Q := First;
                     P := First^.Next;
                     IF Name < P^.Student_name THEN
                        BEGIN
                           NEW (Temp);
                           Temp^.Student_name := Name;
                           Temp^.Social_security := SS_number;
                           Temp^.Next := First^.Next;
                           First^.Next := Temp;
                        END
                      ELSE
                        BEGIN
                           WHILE (Name > P^.Student_name)
                                 AND (P^.Next <> NIL) DO
                              BEGIN
                                 Q := P;
                                 P := Q^.Next;
                              END;
                           IF Name > P^.Student_name THEN
                              BEGIN
                                 NEW (Temp);
                                 Temp^.Student_name := Name;
                                 Temp^.Social_security
                                                  := SS_number;
                                 Temp^.Next := P^.Next;
                                 P^.Next := Temp;
                              END
                            ELSE
                              BEGIN
                                 NEW (Temp);
                                 Temp^.Student_name := Name;
                                 Temp^.Social_security := SS_number;
                                 Temp^.Next := Q^.Next;
                                 Q^.Next := Temp;
                              END
                        END;
                  END;
            END;
         READ (Permfile, Command);
         FOR Column := 1 TO 30 DO
            BEGIN
               READ (Permfile, Ch);
               Name [Column - 1] := Ch;
```

```
            END;
         FOR Column := 37 TO 45 DO
            BEGIN
               READ (Permfile, Ch);
               SS_number [Column - 36] := Ch;
            END;
         READLN (Permfile);
      END;
   CLOSE (Permfile);
   RESET (Transfile);
   READ (Transfile, Command);
   FOR Column := 2 TO 36 DO
      BEGIN
         READ (Transfile, Ch);
         Name [Column - 1] := Ch;
      END;
   FOR Column := 37 TO 45 DO
      BEGIN
         READ (Transfile, Ch);
         SS_number [Column - 36] := Ch;
      END;
   READLN (Transfile);
   WHILE NOT EOF (Transfile) DO
      BEGIN
         CASE Command OF
              'A' : BEGIN
                       IF First .Next = NIL THEN
                          BEGIN
                             NEW (Temp);
                             Temp .Student_name := Name;
                             Temp .Social_security := SS_number;
                             Temp .Next := First .Next;
                             First .Next := Temp;
                          END
                        ELSE
                          BEGIN
                             Q := First;
                             P := First .Next;
                             Found := FALSE;
                             WHILE (P <> NIL) AND (NOT Found) DO
                                IF (P .Student_name = Name) AND
                                   (P .Social_security = SS_number)
                                      THEN Found := TRUE
                                      ELSE
                                         BEGIN
                                            Q := P;
                                            P := P .Next;
                                         END;
                             IF Found THEN
                                BEGIN
                                   WRITELN (Printer);
                                   WRITELN (Printer,
                                              'Duplicate record: Not Added');
                                END
                              ELSE
                                BEGIN
                                   Q := First;
                                   P := First .Next;
                                   IF Name < P .Student_name THEN
                                      BEGIN
                                         NEW (Temp);
                                         Temp .Student_name := Name;
                                         Temp .Social_security := SS_number;
                                         Temp .Next := First .Next;
                                         First .Next := Temp;
                                      END
```

```
                                        ELSE
                                          BEGIN
                                             WHILE (Name <> P^.Student_name)
                                                   AND (P^.Next <> NIL) DO
                                                BEGIN
                                                   Q := P;
                                                   P := Q^.Next;
                                                END;
                                             IF Name < P^.Student_name THEN
                                                BEGIN
                                                   NEW (Temp);
                                                   Temp^.Student_name := Name;
                                                   Temp^.Social_security
                                                               := SS_number;
                                                   Temp^.Next := P^.Next;
                                                   P^.Next := Temp;
                                                END
                                              ELSE
                                                BEGIN
                                                   NEW (Temp);
                                                   Temp^.Student_name
                                                              := Name;
                                                   Temp^.Social_security
                                                              := SS_number;
                                                   Temp^.Next := Q^.Next;
                                                   Q^.Next := Temp;
                                                END
                                          END;
                                    END;
                              END;
                        END;
                  D   : BEGIN
                           Q := First;
                           P := First^.Next;
                           Found := FALSE;
                           WHILE (P <> NIL) AND (NOT Found) DO
                              IF (P^.Student_name = Name) AND
                                 (P^.Social_security = SS_number)
                                THEN Found := TRUE
                                ELSE
                                   BEGIN
                                      Q := P;
                                      P := P^.Next;
                                   END;
                           IF Found THEN
                              Q^.Next := P^.Next
                            ELSE
                              BEGIN
                                 WRITELN (Printer);
                                 WRITELN (Printer,
                                         'Student not in class: ',
                                         'No drop done.');
                              END;
                        END;
                  L   : BEGIN
                           Q := First;
                           P := First^.Next;
                           Found := FALSE;
                           WHILE (P <> NIL) AND (NOT Found) DO
                              IF (P^.Student_name = Name) AND
                                 (P^.Social_security = SS_number)
                                THEN Found := TRUE
                                ELSE
                                   BEGIN
                                      Q := P;
                                      P := P^.Next;
```

```
                        END;
                    IF Found THEN
                       BEGIN
                          WRITELN (Printer);
                          WRITELN (Printer, Name,
                                  ' is in the record. ');
                       END
                     ELSE
                       BEGIN
                          WRITELN (Printer);
                          WRITELN (Printer, Name,
                                  ' is NOT in the record.');
                       END;
                 END;
            L   : BEGIN
                    Q := First;
                    P := First .Next;
                    WRITELN (Printer);
                    WHILE P <> NIL DO
                       BEGIN
                          WRITELN (Printer, P .Student_name,
                                       P .Social_security);
                          P := P .Next;
                       END;
                 END;
         END;
        IF NOT (Command IN ['A', 'D', 'I', 'L']) THEN
           BEGIN
              WRITELN (Printer);
              WRITELN (Printer,
                     'Invalid command; Line from transaction ',
                     'file ignored. ');
           END;
        READ (Transfile, Command);
        FOR Column := 2 TO 36 DO
           BEGIN
              READ (Transfile, Ch);
              Name [Column - 1] := Ch;
           END;
        FOR Column := 37 TO 45 DO
           BEGIN
              READ (Transfile, Ch);
              SS_number [Column - 36] := Ch;
           END;
        READLN (Transfile);
     END;
  CLOSE (Transfile);
  REWRITE (Permfile);
  Q := First;
  P := First .Next;
  WHILE P <> NIL DO
     BEGIN
        WRITELN (Permfile,    ' ', Q .Student_name,
                              Q .Social_security);
        Q := P;
        P := P .Next;
     END;
  CLOSE (Permfile);
  WRITELN (Printer);
  WRITELN (Printer, 'Transaction file completed. ');
  CLOSE (Printer);
  { ** Do NOT alter this line ** }ASSIGN(F, ROW...);REWRITE(F);CLOSE(F);
END.
```

Student Transactions Problem
(Object-Oriented)

```
PROGRAM Classlist (Permfile, Transfile, Printer); {$debug+} {$linesize:132}
{}
{}
{}
{}
     TYPE
          Name_type = PACKED ARRAY [1..35] OF CHAR;
          SS_type = PACKED ARRAY [1..10] OF CHAR;
{}
{}
{}
     VAR
          P : TEXT; { Do NOT alter this line }
          Name : Name_type;
          SS_number : SS_type;
          Column : INTEGER;
          Printer : TEXT;
{}
{}
{}
{}
{}
{}
{ ******   LINKED LIST OBJECT   ********************************}
{}
{}
{}
     TYPE
          Link =  Cell;
{}
          Cell = RECORD
                      Next : Link;
                      Student_name : Name_type;
                      Social_security : SS_type;
                 END;
{}
     VAR [STATIC]
{}
          First, Q, P : Link;
{}
{}
{}
               PROCEDURE Search (VAR Found : BOOLEAN; VAR Q, P : Link);
{}
                    BEGIN
                       Q := First;
                       P := First .Next;
                       Found := FALSE;
                       WHILE (P    NIL) AND (NOT Found) DO
                          IF (P .Student_name = Name) AND
                             (P .Social_security = SS_number)
                                THEN Found := TRUE
                                ELSE
                                   BEGIN
                                      Q := P;
                                      P := P .Next;
                                   END;
                    END;
{}
{}
{}
```

```
   PROCEDURE LIST__Initialize_list;
         BEGIN
            NEW (First);
            First .Next := NIL;
         END;
{}
{}
{}
   PROCEDURE LIST__Add_student;
{}
                    VAR
                       Duplicate : BOOLEAN;
                       X, Y : Link;
{}
                      PROCEDURE Insert_after (After_this : Link);
{}
                                VAR
                                   Temp : Link;
{}
{}
                               BEGIN
                                  NEW (Temp);
                                  Temp .Student_name := Name;
                                  Temp .Social_security := SS_number;
                                  Temp .Next := After_this .Next;
                                  After_this .Next := Temp;
                               END;
{}
{}
                      FUNCTION Empty_list : BOOLEAN;
{}
                               BEGIN
                                  IF First .Next = NIL
                                     THEN Empty_list := TRUE
                                     ELSE Empty list := FALSE;
                               END;
{}
{}
          BEGIN
             IF Empty_list THEN
                Insert_after (First)
              ELSE
                BEGIN
                   Search (Duplicate, X, Y);
                   IF Duplicate THEN
                      Writeln (Printer, 'Duplicate record: Not Added')
                    ELSE
                      BEGIN
                         Q := First;
                         P := First .Next;
                         IF Name   P .Student_name THEN
                            Insert_after (First)
                          ELSE
                            BEGIN
                               WHILE (Name   P .Student_name) AND
                                     (P .Next   NIL) DO
                                  BEGIN
                                     Q := P;
                                     P := Q .Next;
                                  END;
                               IF Name   P .Student_name
                                  THEN Insert_after (P)
                                  ELSE Insert_after (Q);
                            END;
                      END;
                END;
```

```
        END;
{}
{}
{}
{}
  PROCEDURE LIST__Drop_student;
{}
             VAR
                Preceeding, Actual : Link;
                Is_it_there : BOOLEAN;
{}
        BEGIN
           Search (Is_it_there, Preceeding, Actual);
           IF Is_it_there THEN
              Preceeding .Next := Actual .Next
            ELSE
              Writeln (Printer,
                       'Student not in class: No drop done. ');
        END;
{}
{}
{}
{}
      PROCEDURE LIST__Inquire;
{}
                 VAR
                    X, Y : Link;
                    Found : BOOLEAN;
{}
           BEGIN
              Search (Found, X, Y);
              IF Found THEN
                 Writeln (Printer,      , Name,   is in list. )
              ELSE
                 writeln (Printer,      , Name,   is NOT in list. );
           END;
{}
{}
{}
{}
      PROCEDURE LIST__List_all_students;
{}
{}
          BEGIN
             Q := First;
             P := First .Next;
             Writeln (Printer);
             WHILE P    NIL DO
                BEGIN
                   Writeln (Printer,     , P .Student name,
                                 P .Social_security);
                   P := P .Next;
                END;
          END;
{}
{}
     PROCEDURE LIST__Go_to_top_of_list (VAR Not_empty : BOOLEAN);
{}
          BEGIN
             Q := First;
             P := First .Next;
             IF P = NIL
                THEN Not_empty := FALSE
                ELSE Not_empty := TRUE;
          END;
```

```
{}
{}
      PROCEDURE LIST__Get_next_student (VAR Not_empty : BOOLEAN);
{}
            BEGIN
               Name := P^.Student_name;
               SS_number := P^.Social_security;
               P := P^.Next;
               IF P = NIL
                  THEN Not_empty := FALSE
                  ELSE Not_empty := TRUE;
            END;
{}
{}
{}
{}
{ ****************************************************************}
{}
{}
{}
{}
{ ********* Transaction file OBJECT ************************}
{}
{}
{}
         VAR [STATIC]
                  Transfile : TEXT;
                  Command : CHAR;
{}
{}
     PROCEDURE TRANSFILE__Process_trans_file;
{}
                VAR  Added, Dropped, Found : BOOLEAN;
{}
             PROCEDURE Read_data_line;
{}
                   VAR
                        Ch : CHAR;
                        Column : INTEGER;
{}
                   BEGIN
                      READ (Transfile, Command);
                      FOR Column := 2 TO 36 DO
                         BEGIN
                            READ (Transfile, Ch);
                            Name [Column - 1] := Ch;
                         END;
                      FOR Column := 37 TO 45 DO
                         BEGIN
                            READ (Transfile, Ch);
                            SS_number [Column - 36] := Ch;
                         END;
                      READLN (Transfile);
                   END;
{}
          BEGIN
             RESET (Transfile);
             Read_data_line;
             WHILE NOT EOF (Transfile) DO
                BEGIN
                   CASE Command OF
                      'A' : LIST__Add_student;
                      'D' : LIST__Drop_student;
                      'I' : LIST__Inquire;
                      'L' : LIST__List_all_students;
                   END;
```

```
                  IF NOT (Command IN [[illegible]])
                     THEN Writeln (Printer,
                            'Invalid command: line ignored.'[illegible]);
                  Read_data_line;
               END;
         END;
{}
{}
{}
{ ***********************************************************}
{}
{}
{}
{ **********  Permanent FILE OBJECT  ************************}
{}
{}
{}
{}

         VAR [STATIC]
                   Permfile : TEXT;
{}
{}
{}
{}
{}
  PROCEDURE PERMFILE_Read_in_perm_file;
{}
{}
{}
           PROCEDURE Read_perm_file_line;
{}
                  VAR
                     Ch : CHAR;
                     Column : INTEGER;
{}
                  BEGIN
                     READ (Permfile, Ch);
                     FOR Column := 2 TO 36 DO
                        BEGIN
                           READ (Permfile, Ch);
                           Name [Column - 1] := Ch;
                        END;
                     FOR Column := 37 TO 45 DO
                        BEGIN
                           READ (Permfile, Ch);
                           SS_number [Column - 36] := Ch;
                        END;
                     READLN (Permfile);
                  END;
{}
{}
         BEGIN
            RESET (Permfile);
            Read_perm_file_line;
            WHILE NOT EOF (Permfile) DO
               BEGIN
                  LIST_Add_student;
                  Read_perm_file_line;
               END;
            CLOSE (Permfile);
         END;
{}
{}
{}
{}
      PROCEDURE PERMFILE__Save_new_perm_file;
```

```
{}
                    VAR
                        More_left : BOOLEAN;
                        Student_name : Name_type;
                        Social_security : SS_type;
{}
            BEGIN
                REWRITE (Permfile);
                LIST__Go_to_top_of_list (More_left);
                WHILE More_left DO
                    BEGIN
                        LIST__Get_next_student (More_left);
                        WRITELN (Permfile, ' ', Name,
                                                    SS_number);
                    END;
                CLOSE (Permfile);
            END;
{}
{}
{}
{}
{}
{*************************************************************}
{}
{}
{}
{}
{}
{}
{}
BEGIN
    REWRITE (Printer);
    LIST__Initialize_list;
    PERMFILE__Read_in_perm_file;
    TRANSFILE__Process_trans_file;
    PERMFILE__Save_new_perm_file;
    Writeln (Printer);
    Writeln (Printer, 'Transaction file completed. ');
    { ** Do NOT alter this line ** }ASSIGN(F, RUN.OF );REWRITE(F);CLOSE(F);
END.
```

APPENDIX B
PROGRAM OVERVIEWS

PROGRAM OVERVIEW
Host-At-Sea Buoy Problem
Functional Decomposition - Simple

REQUIREMENTS

This program was designed to simulate a real-time system. It concerns a bouy which provides navigation and weather data to air and ship traffic at sea. It collects wind, temperature, and location data, and transmits summaries every 60 seconds, or more detailed information whenever requested by a passing plane or ship. Additionally, in the case of an emergency, it may be told to broadcast an SOS. It will broadcast this SOS every 10 seconds until it is turned off by a separate request. Each bouy has a small computer, 2 temperature sensors (each one at a different depth), wind direction and speed gauges, a location detector, as well as a receiver and a transmitter. Sending an SOS is considered of highest priority, then air and ship requests, respectively, and lastly, the periodic transmissions. To maintain accurate information, readings are taken from the sensing devices at fixed intervals: wind sensors = every 30 secs.; Omega (i.e. location) = every 10 secs; and temperatures = every 10 secs., (5 readings are taken and averaged so to get an accurate determination at each depth). Each sensor reading returns an integer value. Also, the baud rate of data transmission varies depending on whether a ship or plane request was received, due to the time limits of the craft in the vicinity.

DESIGN

This program was broken up into 8 modules. The main process of the program reads in the measurements taken from the five gauges, processes requests received through the receiver and subsequently directs the data to be broadcast by the transmitter. Five of the modules are the processes that take measurements from these gauges. The other two modules are the receiver and the transmitter modules.

MODIFICATION

It has been determined that your wind speed guage is inaccurate. Each time you are asked for the wind speed, read the wind speed guage twice in a row and average the two readings to obtain your reading.

PROGRAM OVERVIEW
Host-At-Sea Buoy Problem
Functional Decomposition - Complex

REQUIREMENTS

This program was designed to simulate a real-time system. It concerns a bouy which provides navigation and weather data to air and ship traffic at sea. It collects wind, temperature, and location data, and transmits summaries every 60 seconds, or more detailed information whenever requested by a passing plane or ship. Additionally, in the case of an emergency, it may be told to broadcast an SOS. It will broadcast this SOS every 10 seconds until it is turned off by a separate request. Each bouy has a small computer, 2 temperature sensors (each one at a different depth), wind direction and speed gauges, a location detector, as well as a receiver and a transmitter. Sending an SOS is considered of highest priority, then air and ship requests, respectively, and lastly, the periodic transmissions. To maintain accurate information, readings are taken from the sensing devices at fixed intervals: wind sensors = every 30 secs.; Omega (i.e. location) = every 10 secs; and temperatures = every 10 secs., (5 readings are taken and averaged so to get an accurate determination at each depth). Each sensor reading returns an integer value. Also, the baud rate of data transmission varies depending on whether a ship or plane request was received, due to the time limits of the craft in the vicinity.

DESIGN

This program was broken up into 8 modules. The main process of the program reads in the measurements taken from the five gauges, processes requests received through the receiver and subsequently directs the data to be broadcast by the transmitter. Five of the modules are the processes that take measurements from these gauges. The other two modules are the receiver and the transmitter modules.

MODIFICATION

If the temperature and wind speed guages have some sort of error (mechanical, electrical), the circuitry associated with it will return the integer 999. If the temperature guage returns 999, you should <u>not</u> count that figure into the average for that averaged reading. (In other words, do not add 999 to the accumulator, and subtract 1 from #_TO_AVG.) If the wind speed guage returns 999, continue reading the guage until you get a reading other than 999.

PROGRAM OVERVIEW
Host-At-Sea Buoy Problem
In-Line - Simple

REQUIREMENTS

This program was designed to simulate a real-time system. It concerns a bouy which provides navigation and weather data to air and ship traffic at sea. It collects wind, temperature, and location data, and transmits summaries every 60 seconds, or more detailed information whenever requested by a passing plane or ship. Additionally, in the case of an emergency, it may be told to broadcast an SOS. It will broadcast this SOS every 10 seconds until it is turned off by a separate request. Each bouy has a small computer, 2 temperature sensors (each one at a different depth), wind direction and speed gauges, a location detector, as well as a receiver and a transmitter. Sending an SOS is considered of highest priority, then air and ship requests, respectively, and lastly, the periodic transmissions. To maintain accurate information, readings are taken from the sensing devices at fixed intervals: wind sensors = every 30 secs.; Omega (i.e. location) = every 10 secs; and temperatures = every 10 secs., (5 readings are taken and averaged so to get an accurate determination at each depth). Each sensor reading returns an integer value. Also, the baud rate of data transmission varies depending on whether a ship or plane request was received, due to the time limits of the craft in the vicinity.

DESIGN

All of the code in this problem is included in the main program. There are no modules, procedures, or functions. It is structured, however, in that it does not contain "GOTO's", but rather controls flow by the use of "while," "repeat... until," "do" loops, etc.

MODIFICATION

It has been determined that your wind speed guage is inaccurate. Each time you are asked for the wind speed, read the wind speed guage <u>twice in a row</u> and average the two readings to obtain your reading.

PROGRAM OVERVIEW
Host-At-Sea Buoy Problem
In-Line - Complex

REQUIREMENTS

This program was designed to simulate a real-time system. It concerns a bouy which provides navigation and weather data to air and ship traffic at sea. It collects wind, temperature, and location data, and transmits summaries every 60 seconds, or more detailed information whenever requested by a passing plane or ship. Additionally, in the case of an emergency, it may be told to broadcast an SOS. It will broadcast this SOS every 10 seconds until it is turned off by a separate request. Each bouy has a small computer, 2 temperature sensors (each one at a different depth), wind direction and speed gauges, a location detector, as well as a receiver and a transmitter. Sending an SOS is considered of highest priority, then air and ship requests, respectively, and lastly, the periodic transmissions. To maintain accurate information, readings are taken from the sensing devices at fixed intervals: wind sensors = every 30 secs.; Omega (i.e. location) = every 10 secs; and temperatures = every 10 secs., (5 readings are taken and averaged so to get an accurate determination at each depth). Each sensor reading returns an integer value. Also, the baud rate of data transmission varies depending on whether a ship or plane request was received, due to the time limits of the craft in the vicinity.

DESIGN

All of the code in this problem is included in the main program. There are no modules, procedures, or functions. It is structured, however, in that it does not contain "GOTO's", but rather controls flow by the use of "while," "repeat... until," "do" loops, etc.

MODIFICATION

If the temperature and wind speed guages have some sort of error (mechanical, electrical), the circuitry associated with it will return the integer 999. If the temperature guage returns 999, you should <u>not</u> count that figure into the average for that averaged reading. (In other words, do not add 999 to the accumulator, and subtract 1 from #_TO_AVG.) If the wind speed guage returns 999, continue reading the guage until you get a reading other than 999.

PROGRAM OVERVIEW
Host-At-Sea Buoy Problem
Object-Oriented - Simple

REQUIREMENTS

This program was designed to simulate a real-time system. It concerns a Host-at-Sea bouy which provides navigation and weather data to air and ship traffic at sea. It collects wind, temperature, and location data, and transmits summaries every 60 seconds, or more detailed information whenever requested by a passing plane or ship. Additionally, in the case of an emergency, it may be told to broadcast an SOS signal every ten seconds; (a separate request will terminate it). Each bouy has a small computer, 2 temperature sensors (each one at a different depth), wind direction and speed gauges, a location detector, as well as a receiver and a transmitter. Sending an SOS is considered of highest priority, then air and ship requests, respectively, and lastly, the periodic transmissions. To maintain accurate information, readings are taken from the sensing devices at fixed intervals: wind sensors = every 30 secs.; Omega (i.e. location) = every 10 secs; and temperatures = every 10 secs., (5 readings are taken and averaged so to get an accurate determination at each depth). Each sensor reading returns an integer value. Also, the baud rate of data transmission varies depending on whether ship or plane request due to time limits of the craft in the vicinity.

DESIGN

This program was broken down into six main sections. The first is GUAGES, which contains all the sensor functions which will read the gauges so measurements can be taken. Second is MEMORY, in which all of the data taken from the gauges that will be later broadcast is stored. RECEIVER accepts current requests for data from passing planes or ships. The TRANSMITTER sends data periodically to any vessel which may be nearby, and sends detailed data or an "SOS" signal, when requested to do so. The fifth section of the program, CLOCK, simulates the passage of time so that the appropriate readings may be taken at the proper intervals. Finally, the MAIN PROCESS controls each of the other sections, beginning them, processing the information which is accummulated in them, processing requests, and directing the transmission of the data stored.

MODIFICATION

It has been determined that your wind speed guage is inaccurate. Each time you are asked for the wind speed, read the wind speed guage twice in a row and average the two readings to obtain your reading.

PROGRAM OVERVIEW
Host-At-Sea Buoy Problem
Object-Oriented - Complex

REQUIREMENTS

This program was designed to simulate a real-time system. It concerns a Host-at-Sea bouy which provides navigation and weather data to air and ship traffic at sea. It collects wind, temperature, and location data, and transmits summaries every 60 seconds, or more detailed information whenever requested by a passing plane or ship. Additionally, in the case of an emergency, it may be told to broadcast an SOS signal every ten seconds; (a separate request will terminate it). Each bouy has a small computer, 2 temperature sensors (each one at a different depth), wind direction and speed gauges, a location detector, as well as a receiver and a transmitter. Sending an SOS is considered of highest priority, then air and ship requests, respectively, and lastly, the periodic transmissions. To maintain accurate information, readings are taken from the sensing devices at fixed intervals: wind sensors = every 30 secs.; Omega (i.e. location) = every 10 secs; and temperatures = every 10 secs., (5 readings are taken and averaged so to get an accurate determination at each depth). Each sensor reading returns an integer value. Also, the baud rate of data transmission varies depending on whether ship or plane request due to time limits of the craft in the vicinity.

DESIGN

This program was broken down into six main sections. The first is GUAGES, which contains all the sensor functions which will read the gauges so measurements can be taken. Second is MEMORY, in which all of the data taken from the gauges that will be later broadcast is stored. RECEIVER accepts current requests for data from passing planes or ships. The TRANSMITTER sends data periodically to any vessel which may be nearby, and sends detailed data or an "SOS" signal, when requested to do so. The fifth section of the program, CLOCK, simulates the passage of time so that the appropriate readings may be taken at the proper intervals. Finally, the MAIN PROCESS controls each of the other sections, beginning them, processing the information which is accummulated in them, processing requests, and directing the transmission of the data stored.

MODIFICATION

If the temperature and wind speed guages have some sort of error (mechanical, electrical), the circuitry associated with it will return the integer 999. If the temperature guage returns 999, you should <u>not</u> count that figure into the average for that averaged reading. (In other words, do not add 999 to the accumulator, and subtract 1 from #_TO_AVG.) If the wind speed guage returns 999, continue reading the guage until you get a reading other than 999.

PROGRAM OVERVIEW
Military Address Problem
Functional Decomposition - Simple

REQUIREMENTS

This program is designed to search for and print the addresses within a certain Postal code area, and/or to do the same for the addresses with- in a certain O-Grade, (the numerical representation of an officer's rank.) It also keeps a running total of the number of labels printed out for each zip code and a breakdown of the number sent to each rank within that zip code. In the database, addresses follow one after the other, each in a separate record, and can be read in as records. The records are sorted by zip code, and, within zip, by grade. Each address consists of 11 fields, each field on one line, which follow sequentially, in the following order: Title, Last Name, Given Names, Branch or Code, Command or Activity, Street or P.O.Box, City, State or Provinvce, Country, Postal code, O-Grade. The output format for labels is: [line 1]Title Given Names Last Name [2]Branch or Code [3]Command or Activity [4]City, State or Province [5]Country Postal Code.

DESIGN OVERVIEW

This program was broken down into 2 primary modules. The first is the data file which contains the records to be examined. The other is the main process which examines the data for matches to the input criteria specified by the user on the terminal.

MODIFICATION

The mailing label currently does not print the street address. The labels should be changed so that the street address appears as the forth line of the label.

EXAMPLE:

Lt. George Smith
Air Force
Bolling
1234 West Street <— this is the new line added
Washington, D.C.
22303

PROGRAM OVERVIEW
Military Address Problem
Functional Decomposition - Complex

REQUIREMENTS

This program is designed to search for and print the addresses within a certain Postal code area, and/or to do the same for the addresses within a certain O-Grade, (the numerical representation of an officer's rank.) It also keeps a running total of the number of labels printed out for each zip code and a breakdown of the number sent to each rank within that zip code. In the database, addresses follow one after the other, each in a separate record, and can be read in as records. The records are sorted by zip code, and, within zip, by grade. Each address consists of 11 fields, each field on one line, which follow sequentially, in the following order: Title, Last Name, Given Names, Branch or Code, Command or Activity, Street or P.O.Box, City, State or Provinvce, Country, Postal code, O-Grade. The output format for labels is: [line 1]Title Given Names Last Name [2]Branch or Code [3]Commmand or Activity [4]City, State or Province [5]Country Postal Code.

DESIGN OVERVIEW

This program was broken down into 2 primary modules. The first is the data file which contains the records to be examined. The other is the main process which examines the data for matches to the input criteria specified by the user on the terminal.

MODIFICATION

The name line currently prints the person's title, given names, and last name (e.g., Lt. Alan C. Schultz). A new data field (a 12th field) is now in the data base, but the program neither recognizes nor uses this information. This field is a Boolean that represents whether or not the person is retired. This field should be incorporated into the program so that this field can be added to the name line as the first item to be printed. With this modification, the output would be as follows:

```
Column:  1234567890123456789012345678901234567890
        If Retired:
          Retired Lt. Alan C. Schultz
        If Not Retired:
          Lt. Alan C. Schultz
```

PROGRAM OVERVIEW
Military Address Problem
In-Line - Simple

REQUIREMENTS

This program is designed to search for and print the addresses within a certain Postal code area, and/or to do the same for the addresses with- in a certain O-Grade, (the numerical representation of an officer's rank.) It also keeps a running total of the number of labels printed out for each zip code and a breakdown of the number sent to each rank within that zip code. In the database, addresses follow one after the other, each in a separate record, and can be read in as records. The records are sorted by zip code, and, within zip, by grade. Each address consists of 11 fields, each field on one line, which follow sequentially, in the following order: Title, Last Name, Given Names, Branch or Code, Command or Activity, Street or P.O.Box, City, State or Provinvce, Country, Postal code, O-Grade. The output format for labels is: [line 1]Title Given Names Last Name [2]Branch or Code [3]Commmand or Activity [4]City, State or Province [5]Country Postal Code.

DESIGN OVERVIEW

This program was written entirely with in-line code such that all code is included in the main program. There are no modules, procedures or functions, although it is structured in that it does not use "goto's", but rather controls flow by the use of "while," "repeat...until," "do" loops, etc.

MODIFICATION

The mailing label currently does not print the street address. The labels should be changed so that the street address appears as the forth line of the label.

EXAMPLE:

```
Lt. George Smith
Air Force
Bolling
1234 West Street        <—— this is the new line added
Washington, D.C.
22303
```

PROGRAM OVERVIEW
Military Address Problem
In-Line - Complex

REQUIREMENTS

This program is designed to search for and print the addresses within a certain Postal code area, and/or to do the same for the addresses with- in a certain O-Grade, (the numerical representation of an officer's rank.) It also keeps a running total of the number of labels printed out for each zip code and a breakdown of the number sent to each rank within that zip code. In the database, addresses follow one after the other, each in a separate record, and can be read in as records. The records are sorted by zip code, and, within zip, by grade. Each address consists of 11 fields, each field on one line, which follow sequentially, in the following order: Title, Last Name, Given Names, Branch or Code, Command or Activity, Street or P.O.Box, City, State or Provinvce, Country, Postal code, O-Grade. The output format for labels is: [line 1]Title Given Names Last Name [2]Branch or Code [3]Command or Activity [4]City, State or Province [5]Country Postal Code.

DESIGN OVERVIEW

This program was written entirely with in-line code such that all code is included in the main program. There are no modules, procedures or functions, although it is structured in that it does not use "goto's", but rather controls flow by the use of "while," "repeat...until," "do" loops, etc.

MODIFICATION

The name line currently prints the person's title, given names, and last name (e.g., Lt. Alan C. Schultz). A new data field (a 12th field) is now in the data base, but the program neither recognizes nor uses this information. This field is a Boolean that represents whether or not the person is retired. This field should be incorporated into the program so that this field can be added to the name line as the first item to be printed. With this modification, the output would be as follows:

```
Column:  1234567890123456789012345678901234567890
        If Retired:
          Retired Lt. Alan C. Schultz
        If Not Retired:
          Lt. Alan C. Schultz
```

PROGRAM OVERVIEW
Military Address Problem
Object-Oriented - Simple

REQUIREMENTS

This program is designed to search for and print the addresses within a certain Postal code area, and/or to do the same for the addresses with- in a certain O-Grade, (the numerical representation of an officer's rank.) It also keeps a running total of the number of labels printed out for each zip code and a breakdown of the number sent to each rank within that zip code. In the database, addresses follow one after the other, each in a separate record, and can be read in as records. The records are sorted by zip code, and, within zip, by grade. Each address consists of 11 fields, each field on one line, which follow sequentially, in the following order: Title, Last Name, Given Names, Branch or Code, Command or Activity, Street or P.O.Box, City, State or Provinvce, Country, Postal code, O-Grade. The output format for labels is: [line 1]Title Given Names Last Name [2]Branch or Code [3]Command or Activity [4]City, State or Province [5]Country Postal Code.

DESIGN OVERVIEW

This program was broken down into three main sections: the file object, which contains the records to be examined; the label object, which formats the information to be printed on the labels; and the main process, which controls all operations on these objects, temporarily stores and passes information, and reads input from the terminal

MODIFICATION

The mailing label currently does not print the street address. The labels should be changed so that the street address appears as the forth line of the label.

EXAMPLE:

Lt. George Smith
Air Force
Bolling
1234 West Street <--- this is the new line added
Washington, D.C.
22303

PROGRAM OVERVIEW
Military Address Problem
Object-Oriented - Complex

REQUIREMENTS

This program is designed to search for and print the addresses within a certain Postal code area, and/or to do the same for the addresses with- in a certain O-Grade, (the numerical representation of an officer's rank.) It also keeps a running total of the number of labels printed out for each zip code and a breakdown of the number sent to each rank within that zip code. In the database, addresses follow one after the other, each in a separate record, and can be read in as records. The records are sorted by zip code, and, within zip, by grade. Each address consists of 11 fields, each field on one line, which follow sequentially, in the following order: Title, Last Name, Given Names, Branch or Code, Command or Activity, Street or P.O.Box, City, State or Provinvce, Country, Postal code, O-Grade. The output format for labels is: [line 1]Title Given Names Last Name [2]Branch or Code [3]Commmand or Activity [4]City, State or Province [5]Country Postal Code.

DESIGN OVERVIEW

This program was broken down into three main sections: the file object, which contains the records to be examined; the label object, which formats the information to be printed on the labels; and the main process, which controls all operations on these objects, temporarily stores and passes information, and reads input from the terminal

MODIFICATION

The name line currently prints the person's title, given names, and last name (e.g., Lt. Alan C. Schultz). A new data field (a 12th field) is now in the data base, but the program neither recognizes nor uses this information. This field is a Boolean that represents whether or not the person is retired. This field should be incorporated into the program so that this field can be added to the name line as the first item to be printed. With this modification, the output would be as follows:

```
Column:  1234567890123456789012345678901234567890
       If Retired:
         Retired Lt. Alan C. Schultz
       If Not Retired:
         Lt. Alan C. Schultz
```

PROGRAM OVERVIEW
Student Transactions Problem
Functional Decomposition - Simple

REQUIREMENTS

This program is designed to update the registrar's listings for students at a university. The registrar has on disk (called the permanent file) the name and social security number of each student enrolled (in alphabetical order). Each day a transaction file is created which contains a command followed by, when needed, the student's name and social security number. The commands are: `A' = add a student in the proper alphabetic location, `D' = drop a student, `I' = inquire about whether a student is enrolled, and `L' = list all students. `A', `D', and `I' require a student name and social security number; `L' does not. The format of the permanent file is: [column 1] blank, [column 2-36] name, [column 37-45] social security number. The format of the transaction file is: [column 1] command, [column 2-36] name, [column 37-45] social security number. In each case, the social security number is written without spaces or hyphens. The program reads the permanent file into a linked list in main memory. It then reads each line of the transactional file and modifies the linked list accordingly. Once the transactional file is finished, the linked list is copied back to the permanent file.

DESIGN

This program was broken down into three primary modules. The first is the permanent file which contains the official list of all students and their social security numbers (in alphabetical order). The second is the transaction file, which consists of all requests of or alteration to the list which need to be done. The third module, the main process, actually performs the operations.

MODIFICATION

The following should be added to the output. When doing the `L' command, count the number of students, and after all the student names have been printed, print the total number of students using the following format:

```
Column 123456789012345678901234567890
               Last name in list
             Total students: *
```

* indicates that the integer value associated with this total should be printed starting in this column.

PROGRAM OVERVIEW
Student Transactions Problem
Functional Decomposition - Complex

REQUIREMENTS

This program is designed to update the registrar's listings for students at a university. The registrar has on disk (called the permanent file) the name and social security number of each student enrolled (in alphabetical order). Each day a transaction file is created which contains a command followed by, when needed, the student's name and social security number. The commands are: `A' = add a student in the proper alphabetic location, `D' = drop a student, `I' = inquire about whether a student is enrolled, and `L' = list all students. `A', `D', and `I' require a student name and social security number; `L' does not. The format of the permanent file is: [column 1] blank, [column 2-36] name, [column 37-45] social security number. The format of the transaction file is: [column 1] command, [column 2-36] name, [column 37-45] social security number. In each case, the social security number is written without spaces or hyphens. The program reads the permanent file into a linked list in main memory. It then reads each line of the transactional file and modifies the linked list accordingly. Once the transactional file is finished, the linked list is copied back to the permanent file.

DESIGN

This program was broken down into three primary modules. The first is the permanent file which contains the official list of all students and their social security numbers (in alphabetical order). The second is the transaction file, which consists of all requests of or alteration to the list which need to be done. The third module, the main process, actually performs the operations.

MODIFICATION

The permanent file now contains some additional information about the class of the student (freshman, sophomore, junior, senior, graduate). This information is contained in column 46 of each record in the permfile as a number in character format.

1 = Freshman
2 = Sophomore
3 = Junior
4 = Senior
5 = Graduate.

Change the `L' command so that when it prints the student list, it prints the number representing class membership immediately following the SS number (i.e. with no spaces between the two.) In making this modification, remember that the program should read in this new information and preserve it for use in the transactions.

```
Column  1234567890123456789012345678901234567890123456789

example:
        Anderson, Harry                     0099811231
                                                     ^
                                                     |
                                                     |
                                            This is the number repre-
                                            senting class membership
```

PROGRAM OVERVIEW
Student Transactions Problem
In-Line - Simple

REQUIREMENTS

This program is designed to update the registrar's listings for students at a university. The registrar has on disk (called the permanent file) the name and social security number of each student enrolled (in alphabetical order). Each day a transaction file is created which contains a command followed by, when needed, the student's name and social security number. The commands are: `A' = add a student in the proper alphabetic location, `D' = drop a student, `I' = inquire about whether a student is enrolled, and `L' = list all students. `A', `D', and `I' require a student name and social security number; `L' does not. The format of the permanent file is: [column 1] blank, [column 2-36] name, [column 37-45] social security number. The format of the transaction file is: [column 1] command, [column 2-36] name, [column 37-45] social security number. In each case, the social security number is written without spaces or hyphens. The program reads the permanent file into a linked list in main memory. It then reads each line of the transactional file and modifies the linked list accordingly. Once the transactional file is finished, the linked list is copied back to the permanent file.

DESIGN

All of the code in this problem is included in the main program. There are no modules, procedures, or functions. It is structured, however, in that it does not contain "GOTO's", but rather controls flow by the use of "while," "repeat... until," "do" loops, etc.

MODIFICATION

The following should be added to the output. When doing the `L' command, count the number of students, and after all the student names have been printed, print the total number of students using the following format:

```
Column 123456789012345678901234567890
              Last name in list
              Total students: *
```

* indicates that the integer value associated with this total should be printed starting in this column.

PROGRAM OVERVIEW
Student Transactions Problem
In-Line - Complex

REQUIREMENTS

This program is designed to update the registrar's listings for students at a university. The registrar has on disk (called the permanent file) the name and social security number of each student enrolled (in alphabetical order). Each day a transaction file is created which contains a command followed by, when needed, the student's name and social security number. The commands are: `A' = add a student in the proper alphabetic location, `D' = drop a student, `I' = inquire about whether a student is enrolled, and `L' = list all students. `A', `D', and `I' require a student name and social security number; `L' does not. The format of the permanent file is: [column 1] blank, [column 2-36] name, [column 37-45] social security number. The format of the transaction file is: [column 1] command, [column 2-36] name, [column 37-45] social security number. In each case, the social security number is written without spaces or hyphens. The program reads the permanent file into a linked list in main memory. It then reads each line of the transactional file and modifies the linked list accordingly. Once the transactional file is finished, the linked list is copied back to the permanent file.

DESIGN

All of the code in this problem is included in the main program. There are no modules, procedures, or functions. It is structured, however, in that it does not contain "GOTO's", but rather controls flow by the use of "while," "repeat... until," "do" loops, etc.

MODIFICATION

The permanent file now contains some additional information about the class of the student (freshman, sophomore, junior, senior, graduate). This information is contained in column 46 of each record in the permfile as a number in character format.

1 = Freshman
2 = Sophomore
3 = Junior
4 = Senior
5 = Graduate.

Change the `L' command so that when it prints the student list, it prints the number representing class membership immediately following the SS number (i.e. with no spaces between the two.) In making this modification, remember that the program should read in this new information and preserve it for use in the transactions.

```
Column  1234567890123456789012345678901234567890123456789O

example:
        Anderson, Harry                   0099811231
                                                   ^
                                                   |
                                                   |
                                        This is the number repre-
                                        senting class membership
```

PROGRAM OVERVIEW
Student Transactions Problem
Object-Oriented - Simple

REQUIREMENTS

This program is designed to update the registrar's listings for students at a university. The registrar has on disk (called the permanent file) the name and social security number of each student enrolled (in alphabetical order). Each day a transaction file is created which contains a command followed by, when needed, the student's name and social security number. The commands are: `A' = add a student in the proper alphabetic location, `D' = drop a student, `I' = inquire about whether a student is enrolled, and `L' = list all students. `A', `D', and `I' require a student name and social security number; `L' does not. The format of the permanent file is: [column 1] blank, [column 2-36] name, [column 37-45] social security number. The format of the transaction file is: [column 1] command, [column 2-36] name, [column 37-45] social security number. In each case, the social security number is written without spaces or hyphens. The program reads the permanent file into a linked list in main memory. It then reads each line of the transactional file and modifies the linked list accordingly. Once the transactional file is finished, the linked list is copied back to the permanent file.

DESIGN

This program was broken down into four main sections. The first is the permanent file object, which contains the official list of all students and their social security numbers (in alphabetical order). The second is the transaction file object, which consists of all requests of or alteration to the list which need to be done. The third section, the linked list object, is a representation of all students within the computer memory and which is acted upon by the transaction file. And finally, the printer object outputs any requested information, error messages, and a completion message once the transaction file has been successfully processed.

MODIFICATION

The following should be added to the output. When doing the `L' command, count the number of students, and after all the student names have been printed, print the total number of students using the following format:

```
Column 123456789012345678901234567890
        Last name in list
       Total students: *
```

* indicates that the integer value associated with this total should be printed starting in this column.

PROGRAM OVERVIEW
Student Transactions Problem
Object-Oriented - Complex

REQUIREMENTS

This program is designed to update the registrar's listings for students at a university. The registrar has on disk (called the permanent file) the name and social security number of each student enrolled (in alphabetical order). Each day a transaction file is created which contains a command followed by, when needed, the student's name and social security number. The commands are: `A' = add a student in the proper alphabetic location, `D' = drop a student, `I' = inquire about whether a student is enrolled, and `L' = list all students. `A', `D', and `I' require a student name and social security number; `L' does not. The format of the permanent file is: [column 1] blank, [column 2-36] name, [column 37-45] social security number. The format of the transaction file is: [column 1] command, [column 2-36] name, [column 37-45] social security number. In each case, the social security number is written without spaces or hyphens. The program reads the permanent file into a linked list in main memory. It then reads each line of the transactional file and modifies the linked list accordingly. Once the transactional file is finished, the linked list is copied back to the permanent file.

DESIGN

This program was broken down into four main sections. The first is the permanent file object, which contains the official list of all students and their social security numbers (in alphabetical order). The second is the transaction file object, which consists of all requests of or alteration to the list which need to be done. The third section, the linked list object, is a representation of all students within the computer memory and which is acted upon by the transaction file. And finally, the printer object outputs any requested information, error messages, and a completion message once the transaction file has been successfully processed.

MODIFICATION

The permanent file now contains some additional information about the class of the student (freshman, sophomore, junior, senior, graduate). This information is contained in column 46 of each record in the permfile as a number in character format.

1 = Freshman
2 = Sophomore
3 = Junior
4 = Senior
5 = Graduate.

Change the `L' command so that when it prints the student list, it prints the number representing class membership immediately following the SS number (i.e. with no spaces between the two.) In making this modification, remember that the program should read in this new information and preserve it for use in the transactions.

```
Column  1234567890123456789012345678901234567890123456789

example:
        Anderson, Harry                 0099811231
                                                 |
                                        This is the number representing
                                        class membership
```

TECHNICAL REPORTS DISTRIBUTION LIST

OFFICE OF NAVAL RESEARCH

Engineering Psychology Program

TECHNICAL REPORTS DISTRIBUTION LIST

OSD

CAPT Paul R. Chatelier
Office of the Deputy Under Secretary of Defense
OUSDRE (E&LS)
Pentagon, Room 3D129
Washington, DC 20301

Department of the Navy

Engineering Psychology Program
Office of the Naval Research
Code 1142EP
800 North Quincy Street
Arlington, VA 22217-5000 (3 copies)

Dr. Randall P. Schumaker
NRL A.I. Center
Code 7510ical R&D Command
Naval Research Laboratory
Washington, DC 20375-5000

Special Assistant for Marine Corps Matters
Code OOMC
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217-5000

Human Factors Department
Code N-71
Naval Training Systems Center
Orlando, FL 32813

Director
Technical Information Division
Code 2627
Naval Research Laboratory
Washington, DC 23075-5000

Dr. Michael Melich
Communications Sciences Division
Code 7500
Naval Research Laboratory
Washington, DC 23075-5000

Information Sciences Division
Code 1133
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217-5000

CDR T. Jones
Code 125
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217-5000

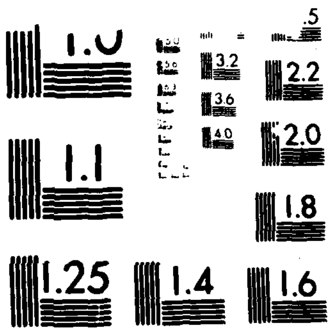
Mr. John Davis
Combat Control Systems Department
Code 35
Naval Underwater Systems Center
Newport, RI 02840

CDR James Offutt
Office of the Secretary of Defense
Strategic Defense Initiative
Organization
Washington, DC 20301-7100

Mr. Norm Beck
Combat Control Systems Department
Code 35
Naval Underwater Systems Center
Newport, RI 02840

Human Factors Engineering
Code 441
Naval Ocean Systems Center
San Diego, CA 92152

AD-A168 775 THE ROLE OF PROGRAM STRUCTURE IN SOFTWARE MAINTENANCE 2/2
(U) GEORGE MASON UNIV FAIRFAX VA DEPT OF PSYCHOLOGY
D A BOEHM-DAVIS ET AL. 29 MAY 86 TR-86-GMU-P01
UNCLASSIFIED N00014-85-K-0243 F/G 9/2 NL


1.0
3.2
3.6
4.0
2.2
2.0
1.1
1.8
1.25
1.4
1.6

Department of the Navy

Dr. Neil McAlister
Office of Chief of Naval Operations
Command and Control
OP-094H
Washington, DC 20350

Dr. Gary Poock
Operations Research Department
Naval Postgraduate School
Monterey, CA 93940

Dr. L. Chmura
Computer Sciences & Systems
Code 7592
Naval Research Laboratory
Washington, DC 20375-5000

Dr. Stanley Collyer
Office of Naval Technology
Code 222
800 North Quincy Street
Arlington, VA 22217-5000

Mr. Philip Andrews
Naval Sea Systems Command
NAVSEA 61R
Washington, DC 20362

Dr. George Moeller
Human Factors Engineering Branch
Naval Submarine Base
Submarine Medical Research Laboratory
Groton, CT 06340

Mr. Jeff Grossman
Human Factors Division, Code 71
Navy Personnel R & D Center
San Diego, CA 92152-6800

Dean of the Academic Departments
US Naval Academy
Annapolis, MD 21402

Human Factors Branch
Code 3152
Naval Weapons Center
China Lake, CA 93555

Dr. Steve Sacks
Naval Electronics Systems Command
Code 61R
Washington, DC 20363-5100

Dr. A. F. Norcio
Computer Sciences & Systems
Code 7592
Naval Research Laboratory
Washington, DC 20375-5000

Dr. A.L. Slafkosky
Scientific Advisor
Commandant of the Marine Corps
Washington, DC 20380

CDR C. Hutchins
Code 55
Naval Postgraduate School
Monterey, CA 93940

Commander
Naval Air Systems Command
Crew Station Design
NAVAIR 5313
Washington, DC 20361

Aircrew Systems Branch
Systems Engineering Test
Directorate
US Naval Test Center
Patuxent River, MD 20670

Dr. Robert Blanchard
Code 71
Navy Personnel Research and
Development Center
San Diego, CA 92152-6800

LT Dennis McBride
Human Factors Branch
Pacific Missle Test Center
Point Mugu, CA 93042

CDR W. Moroney
Naval Air Development Center
Code 602
Warminster, PA 18974

Dr. Eugene E. Gloye
ONR Detachment
1030 East Green Street
Pasadena, CA 91106-2485

Dr. Robert A. Fleming
Human Factors Support Group
Naval Personnel R & D Center
1411 South Fern Street
Arlington, VA 22202

Department of the Army

Dr. Edgar M. Johnson
Technical Director
US Army Research Institute
Alexandria, VA 22333-5600

Technical Director
US Army Human Engineering Lab
Aberdeen Proving Ground, MD 21005

Director
Organizations & Systems Research Lab
US Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333-5600

Dr. Milton S. Katz
Director, Basic Research
Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333-5600

Department of the Air Force

Dr. Kenneth R. Boff
AF AMRL/HE
Wright-Patterson AFB, OH 45433

Mr. Charles Bates, Director
Human Engineering Division
USAF AMRL/HES
Wright-Patterson AFB, OH 45433

Dr. Earl Alluisi
Chief Scientist
AFHRL/CCN
Brooks Air Force Base, TX 78235

Dr. Kenneth Gardner
Applied Psychology Unit
Admiralty Marine Tech. Estab.
Teddington, Middlesex TW11 OLN
ENGLAND

Other Government Agencies

Dr. M.C. Montemerlo
Information Sciences & Human Factors
Code RC
NASA HQS
Washington, DC 20546

Dr. Clinton Kelly
Defense Advanced Research
Projects Agency
1400 Wilson Blvd.
Arlington, VA 22209

Defense Technical Information Center
Cameron Station
Bldg. 5
Alexandria, VA 22314 (12 copies)

Other Organizations

Dr. Jesse Orlansky
Institute for Defense Analyses
1801 N. Beauregard Street
Alexandria, VA 22311

Dr. Scott Robertson
Catholic University
Department of Psychology
Washington, DC 20064

Dr. Stanley Deutsch
NAS-National Research Council
(COHF)
2101 Constitution Avenue, NW
Washington, DC 20418

Ms. Denise Benel
Essex Corporation
333 N. Fairfax Street
Alexandria, VA 22314

Dr. H. McI. Parsons
Essex Corporation
333 N. Fairfax Street
Alexandria, VA 22314

Dr. Marvin Cohen
Decision Science Consortium, Inc.
Suite 721
7700 Leesburg Pike
Falls Church, VA 22043

Dr. William B. Rouse
School of Industrial & Systems Engineering
Georgia Institute of Technology
Atlanta, GA 30332

Dr. Bruce Hamill
The Johns Hopkins University
Applied Physics Lab
Laurel, MD 20707

Dr. Richard Pew
Bolt Beranek & Newman, Inc.
50 Moulton Street
Cambridge, MA 02238

END
DTIC
7-86