1986

Structured software - designed and maintenance, C.E. 309 Lecture Notes, Part II, August 1986

Celal N. Kostem

STRUCTURED SOFTWARE DESIGN AND MAINTENANCE

(CE 309: COMPUTER PROGRAMMING LECTURE NOTES; PART-II)

by

Celal N. Kostem

Fritz Engineering Laboratory

Department of Civil Engineering

Lehigh University

Bethlehem, Pennsylvania

August, 1986

Fritz Engineering Laboratory Report No. 400.31

Table of Contents

FOREWORD

The first part of the CE 309: Computer Programming lecture notes covers the salient FORTRAN77 statements, their proper use in structured programming environment, and the interpretation of frequently used civil engineering formulae, specifications, and analysis techniques in terms of FORTRAN77. The second part, i.e. these lecture notes, cover the key issues in the development, maintenance, and testing of software packages, with emphasis on those coded in FORTRAN77.

The majority of the material in this note set, especially the "lists," is based on the material included in Software Maintenance - The Problem and Its Solutions, by James Martin and Carma McClure (Prentice-Hall, Inc., 1983). Even though this is an excellent source of information on software maintenance, the incorporation of the concepts presented in the coding stage will result in a highly desirable product.

1. STRUCTURED AND MODULAR PROGRAMMING

Engineers have been writing "programs" in FORTRAN for digital computers since 1956. The programs coded in the late fifties and early sixties, even in the late sixties, which used the prevailing programming "styles," would be "unacceptable" if current standards are applied. The American National Standards Institute (ANSI) provided standards for FORTRAN in 1966. This FORTRAN is known as FORTRAN-IV or FORTRAN-66. In the sixties and early seventies the work carried out by the computer scientists led to the development of stringent programming guidelines and to new programming languages. The programming language Pascal, introduced in 1971, is probably the most influential language as far as its effects on today's FORTRAN. In addition, to "remedy" the shortcomings of the earlier programming styles, which were strictly programmer-dependent, in the seventies new concepts, rules, guidelines, and standards emerged. They are: program modules and modular programming, top-down programming, and structured programming. These guidelines were in conflict with the loosely "structured," or unstructured, nature of FORTRAN-66. ANSI issued the standards for FORTRAN-77, which is a highly "structured" language as compared to FORTRAN-66, but which is not structured enough as compared to, for example, Pascal.

1.1 NEED FOR A STRUCTURED LANGUAGE

The best justification for a structured programming style can be noted by studying the fictitious program segment shown in Fig. 1.1. The program is incomprehensible because of the crisscrossing arrows highlighting the transfer of control, or logic. It should also be recognized that this is an extreme example; not all FORTRAN-66 codes were written in such a chaotic manner.

There are a number of reasons given by the proponents of "structured programming" as to why it is the only approach that should be rigidly adhered to (rather dogmatic!). Some of the reasons are:

* **The program must be <u>understandable</u> and <u>comprehensible</u>.**

(Inspection of Fig. 1.1 does not give a clue as to what is happening. This is due to the maze of arrows showing the transfer of control. In a well structured program a brief inspection should easily reveal what is happening.)

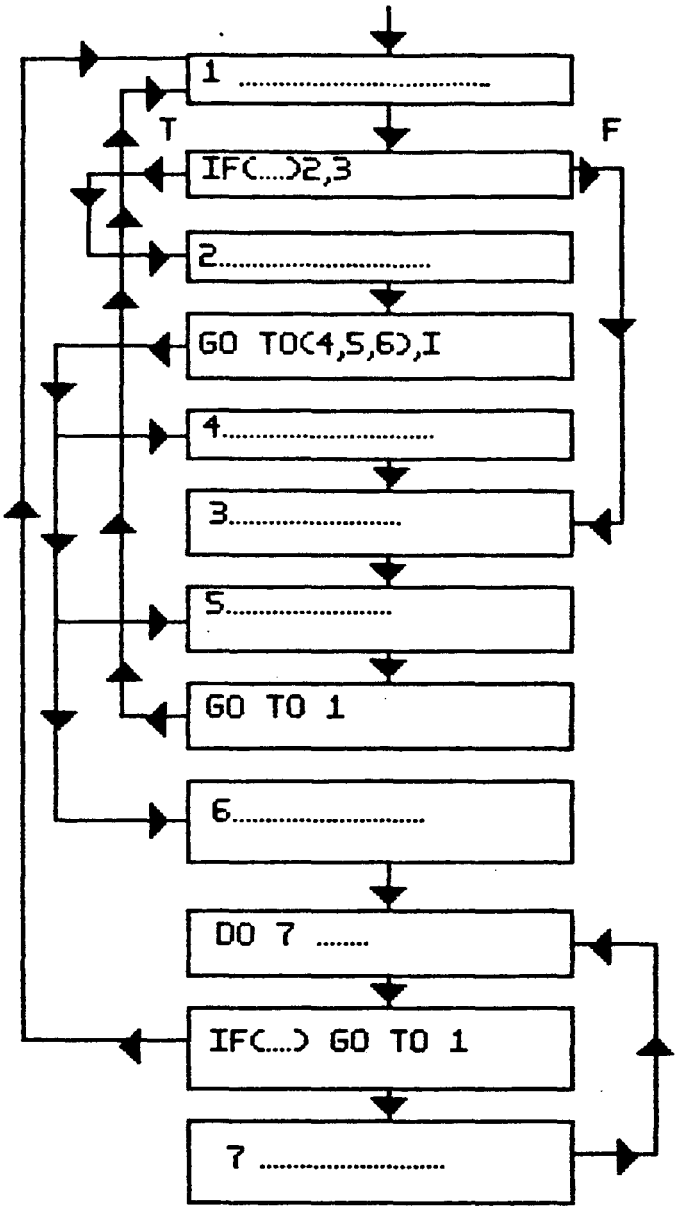


Fig. 1.1 A Sample Nonstructured Program

* The control of transfer of operations should be as sequential as possible.

(This would permit the study and understanding of the program from the top of the page to the bottom, without going back to earlier lines too many times. This also makes the automatic optimization easier.)

* The program should be modular.

(It should be possible to identify the basic self-contained computational and/or other logical units. It should be possible to pull out the module and replace it by another unit serving the same purpose, without making "any(?)" changes in the rest of the program. This feature improves the understandability and modifiability of the software. Furthermore, it is a blessing as far as debugging is concerned.)

* The program must be modifiable.

(In order to modify the program one needs to know what is being done and where, and how it is done. If the program is too rigid, a la Fig. 1.1, its modifiability is in question.)

* The program should be flowchartable.

(Any and every program is flowchartable. Thus, this requirement, prescribed by the experts, is misleading. However, if the logic flow of the program is too complex (polite way of saying haphazard, disorganized, etc.) its flowchart would be almost impossible to follow. Thus the ease of understandability of the flowchart is a requirement for the overall program behavior.)

*Etc.

1.1.1 Counter Arguments

A few case studies observed personally, and lived through, by the author will be presented herein to highlight the possible misunderstandings regarding unstructured vs. structured programming.

CASE-1: A package developed in the late sixties employed standard FORTRAN66. Every care and effort was put forth to make the package insensitive to possible changes in operating system and the compiler. With the emergence of STRUCTURED FORTRAN, a reputable computer scientist indicated that this package should be re-coded for increased efficiency. This mission was undertaken by this scientist. The resulting program did not perform any faster. Thus, not all packages coded in FORTRAN66 can and should be labeled inefficient.

CASE-2: A parallel development was undertaken by a "computer honcho" for the above package in the same time frame as the original development. Use of every imaginable feature of the operating system and the compiler available were made in order to reduce the central core requirement of the computer and increase its execution speed. The finished product performed admirably, both in terms of speed and core requirements. However, every major update in the compiler or the operating system required rewriting parts of the program. Finally, to strip the program of its compiler and operating system redundancy, i.e. to make it fully portable at the expense of losing some of the efficiency, slightly less than 5-minutes of conversion effort was required per statement.

> **Program portability is the most critical attribute of any given program to make it operable in continually changing hardware and software environments.**

CASE-3: In the late sixties and early seventies a major program was developed using FORTRAN66. Prior to the initiation of the activities it was decided that the main program, i.e. primary module, would not be more than 132-lines long. During the development program specifications and requirements were subjected to one major change every month, on the average. About half a dozen major patches were required to be added every three months. At the completion of the programming, debugging, and testing the length of the main program was in excess of 2,000 lines long.

A structured program development was subjected to similar requirements by individuals who are experts on the subject matter but who do not have any idea about software engineering.

> ***Whether a program is coded in structured or an unstructured language, external requirements that can be imposed by the uninitiated can hinder the successful completion of any given package.***

1.2 LIFECYCLE OF SOFTWARE

Activities pertaining to the development of any given software from the definition of the problem statement and/or requirements through the installation of the production version of the software are presented in Fig. 1.2. One can and should be able to go to any one of the activities, and be able to initiate productive efforts. If the developed software can be comprehended only by its originator, so long as he/she can remember what was done, than this software is doomed to oblivion. Any package that is developed with structuredness and modularity can be revived less painfully as compared to the one that does not have these characteristics.

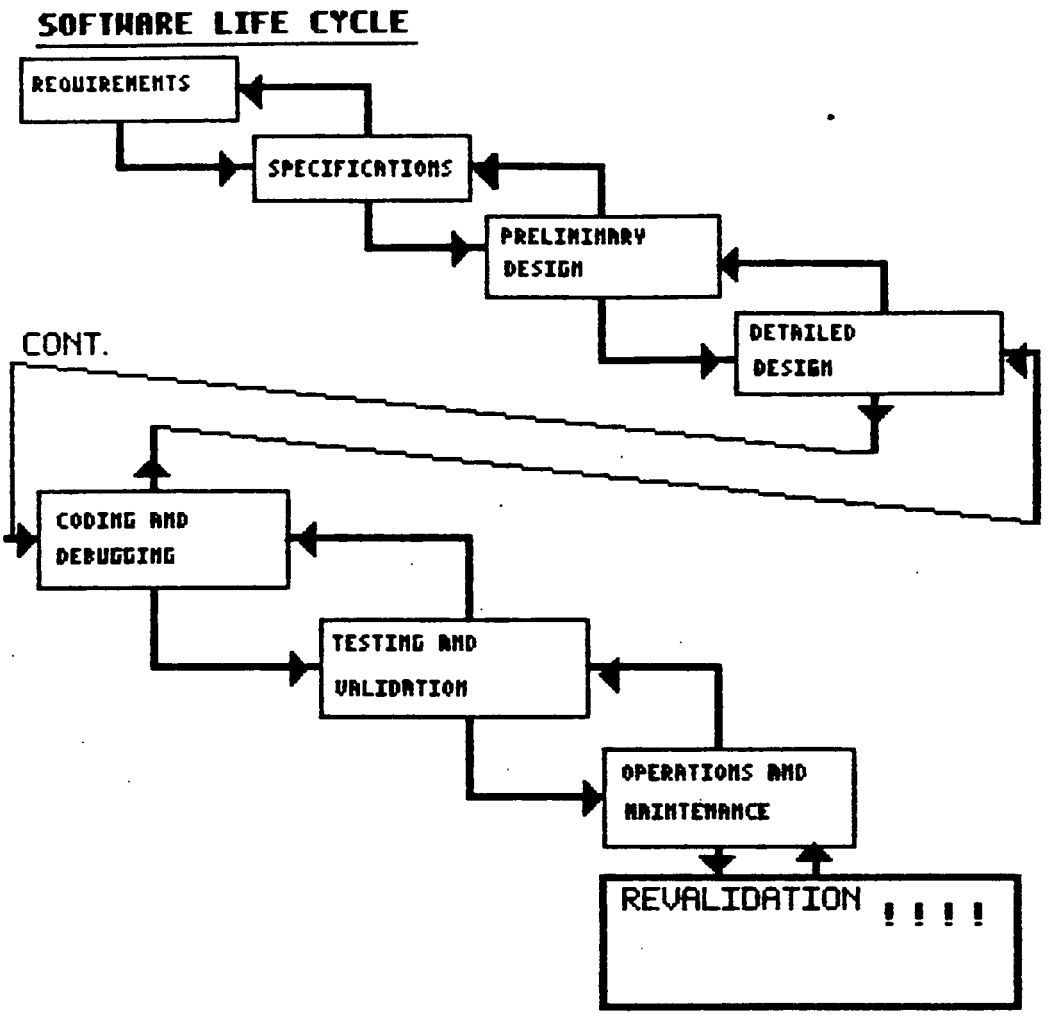


Fig. 1.2 Software Life Cycle

EFFORT REQUIRED ON SOFTWARE DEVELOPMENT

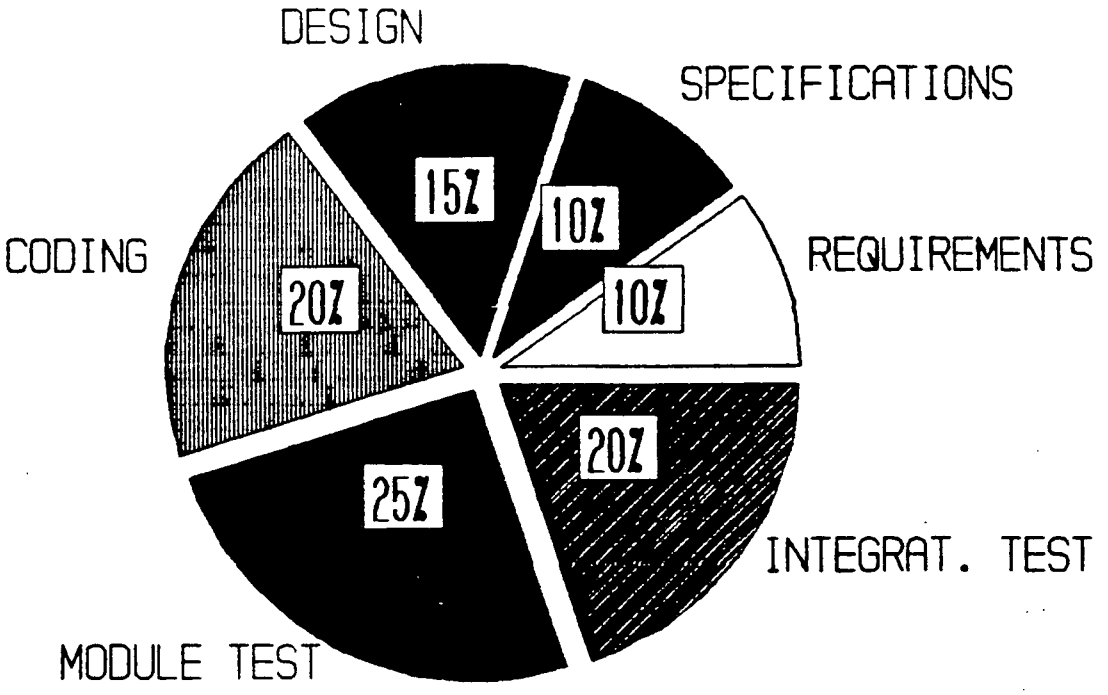


Fig. 1.3 Effort Required on Various Software Development Activities (Except Software Maintenance (from: Principles of Software Engineering and Design, M.A.Zelkowitz, A.C. Shaw, J. D. Gannon, Prentice-Hall, Inc., 1979)

Contrary to beliefs, the time requirement for the coding and debugging is only 20% of the overall effort (See Fig. 1.3). An equal amount of time is spent on the development of the "specifications and requirements" of the software module. Just the design of the software requires 15% of the total effort. Before the inception of any coding effort one should spend about one third of the allocated time and funding.

The percentages given herein are to illustrate the extensive planning required prior to the coding of any software. Actually similar rules of thumb exist for the idealization of data design prior to database-related activities, and/or conceptualization of a problem prior to the use of a canned package. This presupposes that the user is sufficiently familiar with the subject matter. One of the worst practices by engineering managers, especially at the middle-management level, has been to direct the engineers to start "getting computer results" without any initial planning. "Saving" the first 20%-30% of the effort does not reduce the total cost of the project. In the coding, testing and implementation phases major stumbling blocks are usually encountered. This requires a double take, and requires the repetition of previous efforts.

1.3 TOP-DOWN PROGRAMMING

If the "top-down programming" concept is literally applied, the program execution will be a sequential one, starting with the first executable statements until the "STOP" or "RETURN" is encountered. Exception to this rule is the DO-loops. A program like this will not require any "statement numbers," since the control of the execution will not be "controlled." Even though such compilers exist, they are rarely favored by engineers. Actually, most FORTRAN-77 compilers contain extensive enhancements to ANSI FORTRAN77 provisions. Thus, most compilers permit movement to an earlier statement in the program, even though it violates the spirit of the "top-down" practice.

> ***In programming every effort should be put forth to make the program top-down, so long as this does not result in an absurd code.***

1.4 MODULAR PROGRAMMING

A program module is defined as a logically self-contained unit of a program with specific entry and specific exit points, and is aimed to accomplish a predefined mission. Each subroutine can be considered a module, so long as its mission is to perform one task.

Inspection of Fig. 1.4 shows that a part of the program can be isolated as a module, like the IF-THEN-ELSE block. Without knowing the contents of the blocks for the .TRUE. and .FALSE.

branches of the logical IF-check, further suggestions for other sub-modules would not be proper.

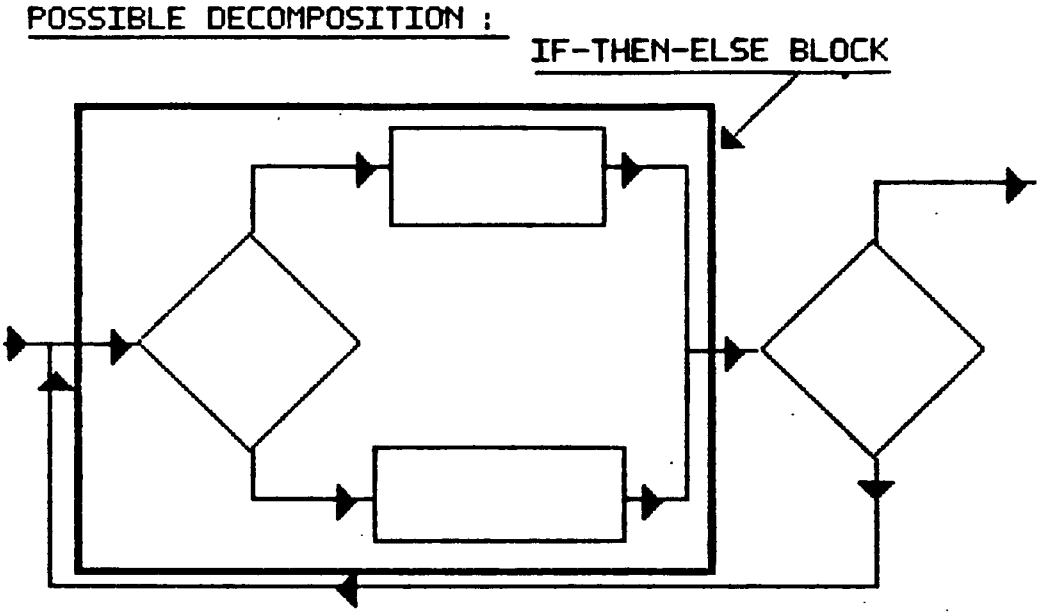


Fig. 1.4 Decomposition/Isolation of a Program Module

1.4.1 Rules for Modularization

Once the basic module concept is understood, then the best guideline for the establishment of modules and modular design is common sense. The following list gives additional guidelines to be noted for modular software design:

1) Decompose the program into independent, discrete modules.

2) Structure the program modules to reflect the design process.

3) Construct each program module with the following properties:

a) The module is closed.

b) The module has one unique entry point and one unique exit point.

c) The module represents one logical, self-contained function.

1.5 STRUCTURED PROGRAMS

The structured programs essentially follow the "top-down" concepts; preferably not too rigidly for the sake of the programmers sanity in transition from FORTRAN66 to FORTRAN77. Ideally these programs do not have any GOTO statements, thereby eliminating the statement labels, i.e. statement numbers. Conversely, a program without GOTO statements, which violates the long list of requirements described in this chapter, would not be a structured program.

1.5.1 Basic Building Blocks

Strictly speaking, the structured program will consist of the building blocks described in Fig. 1.5. So far there has not been a clear cut definition of the terminology for these blocks. The proper terminology for the second block should be "decision." The third block could possibly be labeled as a "DO-loop" as well.

Limited deviation from these blocks, depending upon the options and capabilities of a given compiler, may rarely result in the improvement of efficiency of the program, and may also make it more understandable. Any deviation should not violate the portability of the software; the consequences of which are presented in this document.

Proper composition of the basic building blocks is shown in Fig. 1.6. If each basic building block can be considered as the lowest form of a module, proper composition of the blocks should lead to an acceptable format. Improper use of the modules can be seen in Fig. 1.7. In the first case there are two entry points to the segment. From the description of the modular construction, it is known that there should be only one entry and one exit. The second case is improper because of multiple exits. The third case is improper because of the improper entry to the module. In the top-down approach, it is recommended to enter at the top, and exit at the bottom. The fourth case looks improper because of the relative complexity of the flowchart. This module should be decomposed into two modules.

1.6 PROPERTIES OF A WELL-STRUCTURED PROGRAM

1. The program is divided into a set of modules arranged in a hierarchy defining their logical and execution-time relationships.

2. The execution flow from module to module is restricted

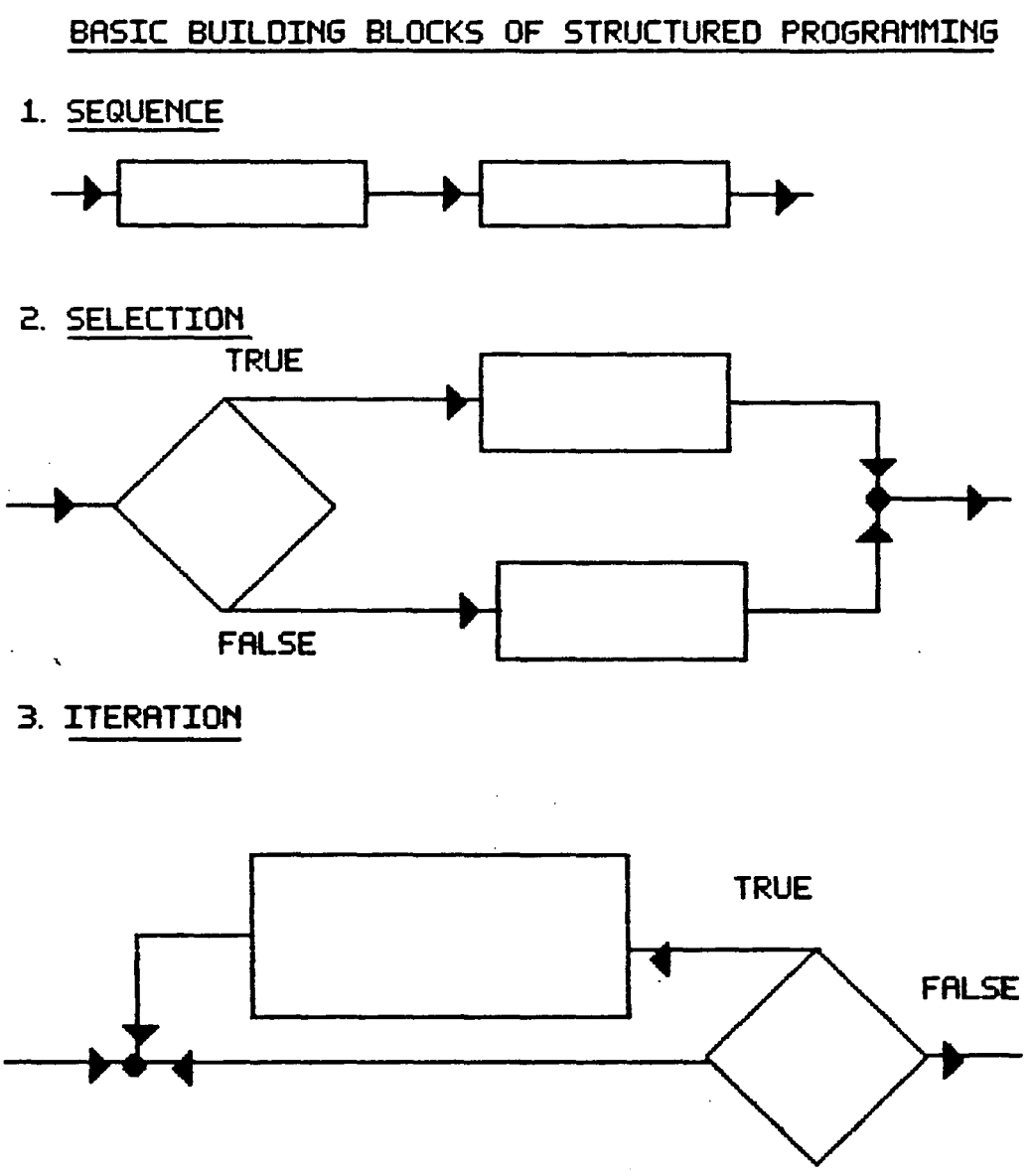


Fig. 1.5 Basic "Building Blocks" of Structured Programming

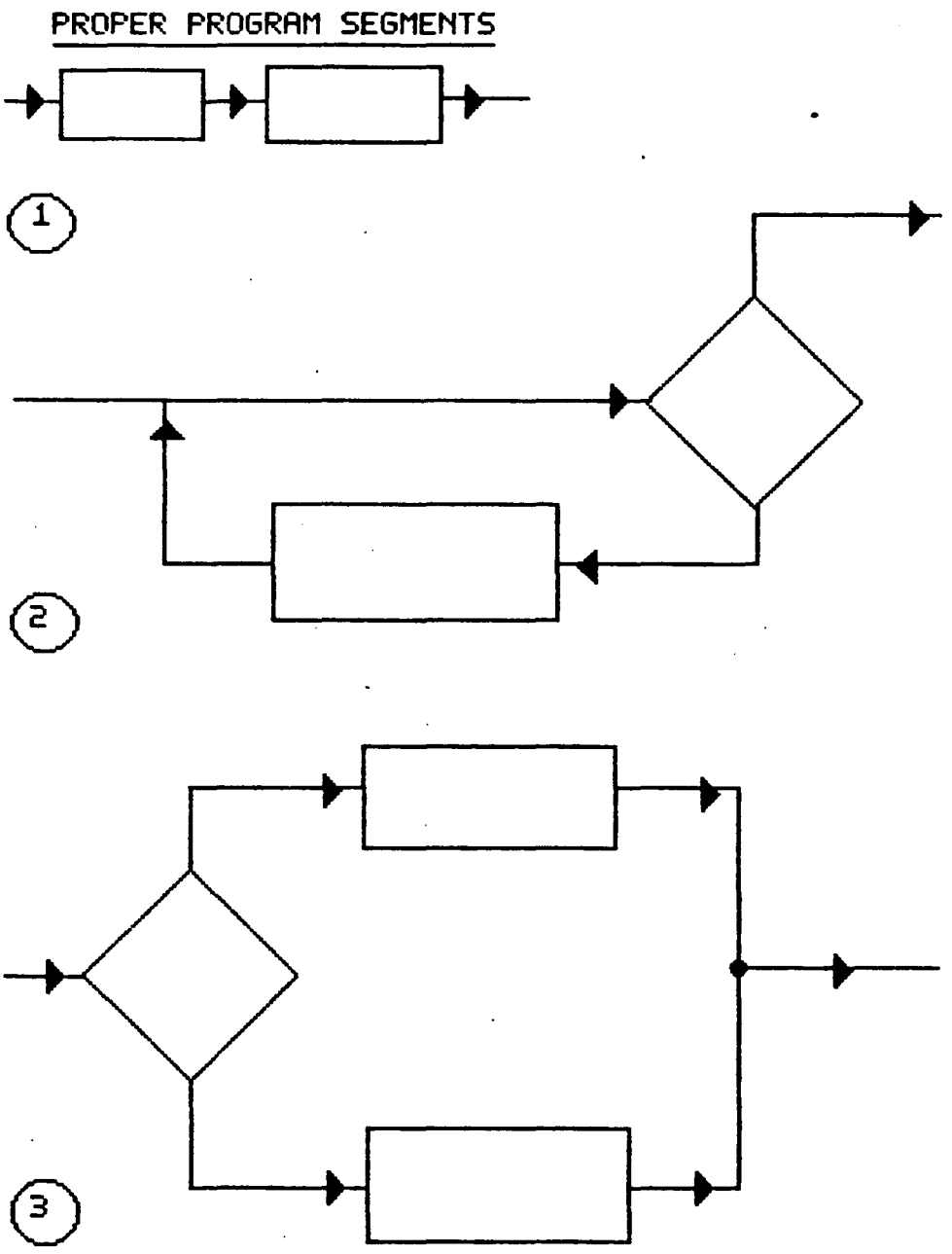


Fig. 1.6 Integration of Basic Building Blocks (Modular Construction)

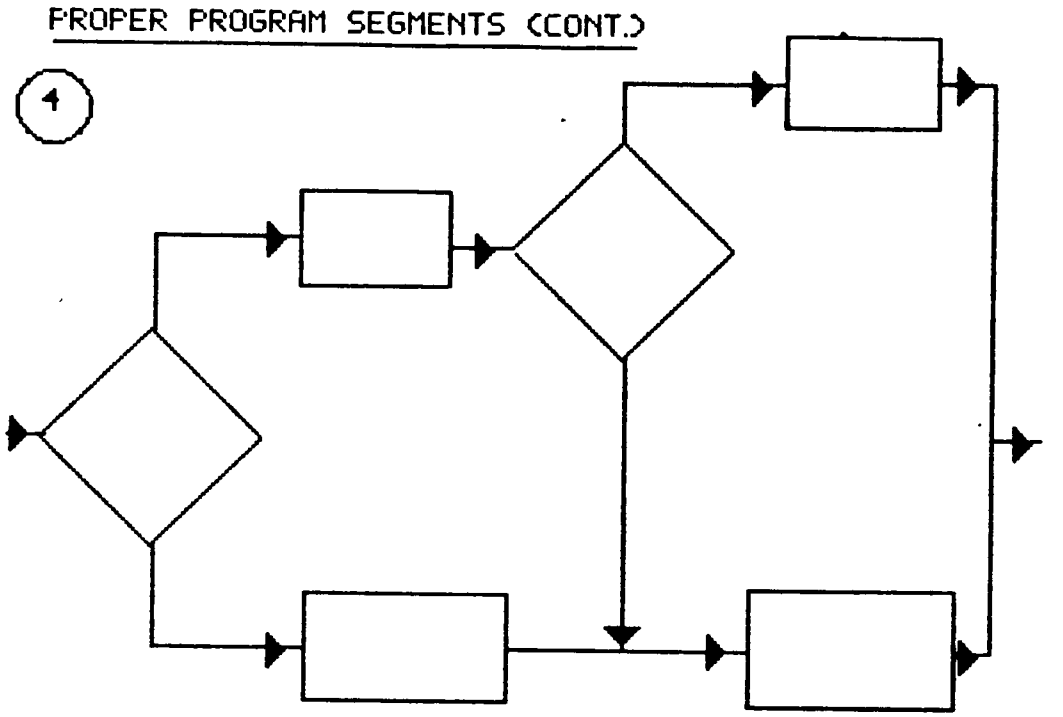


Fig. 1.6 (Cont.) Integration of Basic Building Blocks (Modular Construction)

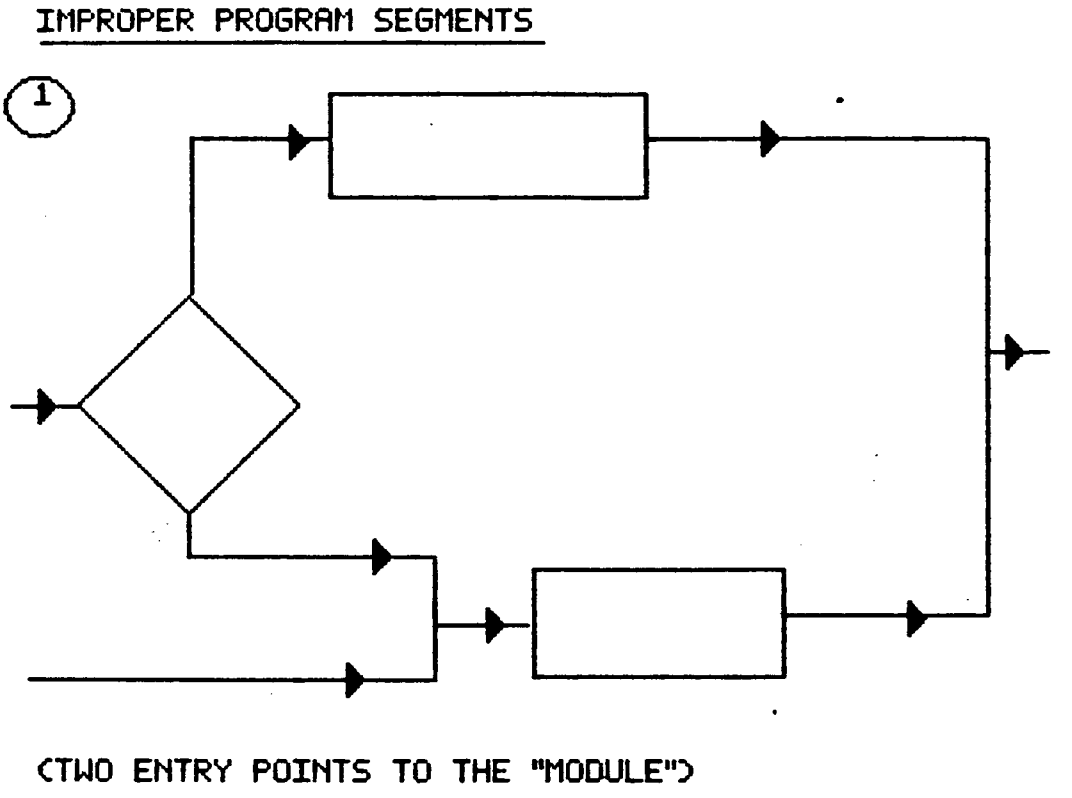
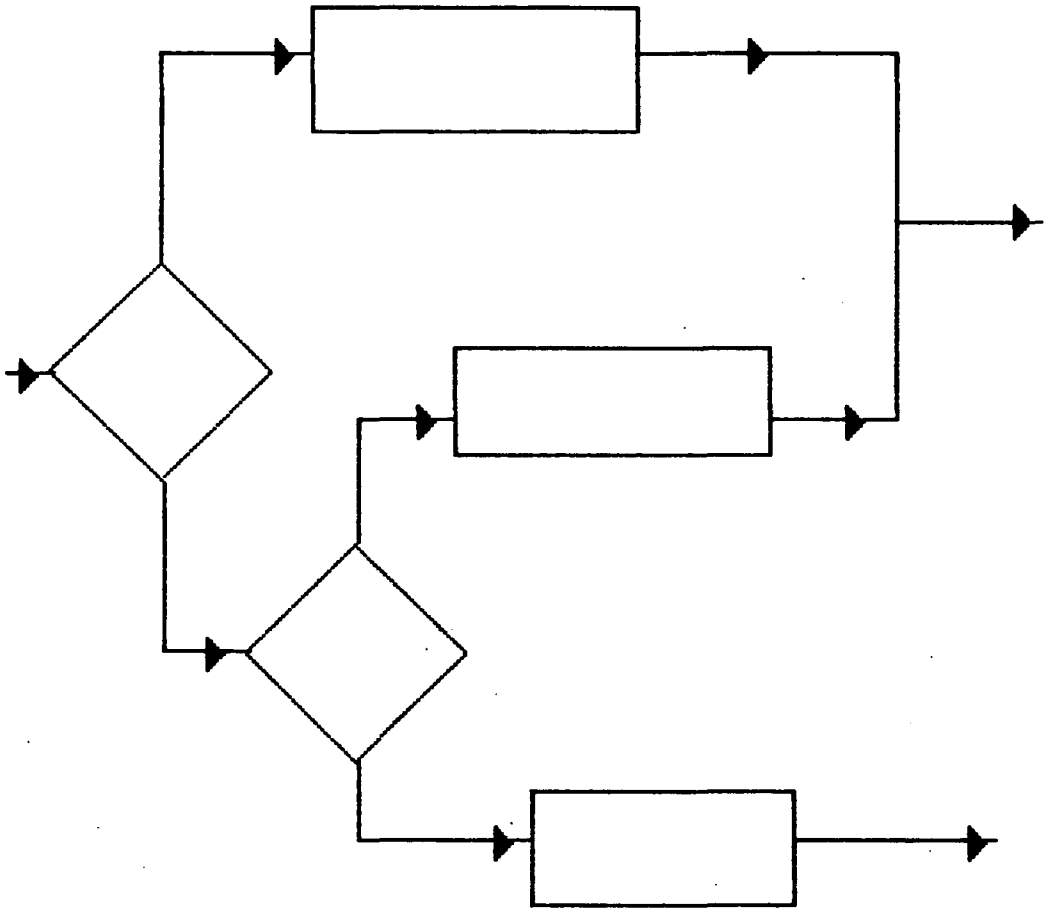


Fig. 1.7 Improper "Integration" of Basic Building Blocks

IMPROPER PROGRAM SEGMENTS

(2) (TWO EXIT POINTS FROM THE MODULE.)



IMPROPER PROGRAM SEGMENTS

("MIDPOINT" ENTRY TO THE MODULE.)

(3)

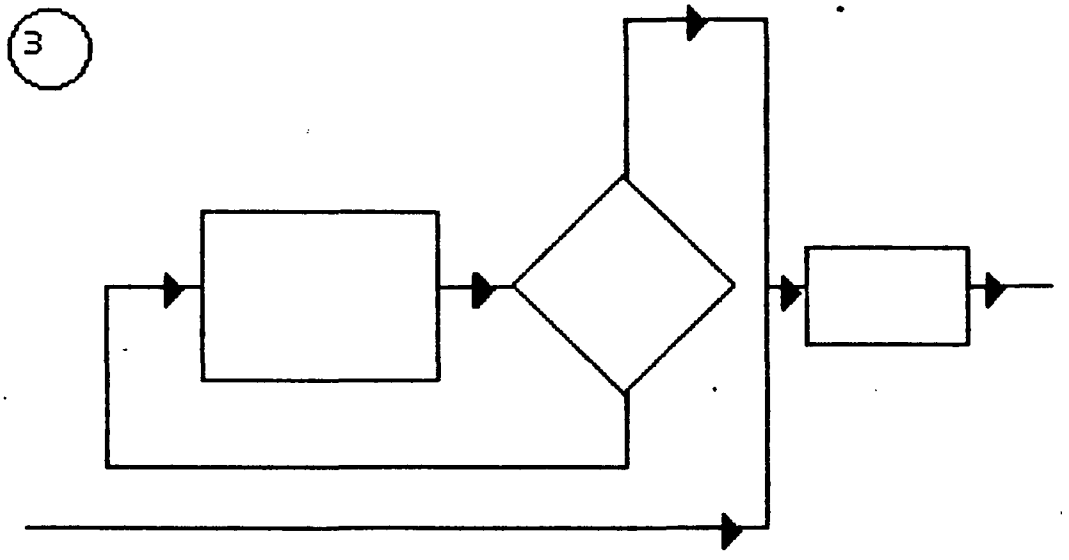


Fig. 1.7 (Cont.) Improper "Integration" of Basic Building Blocks

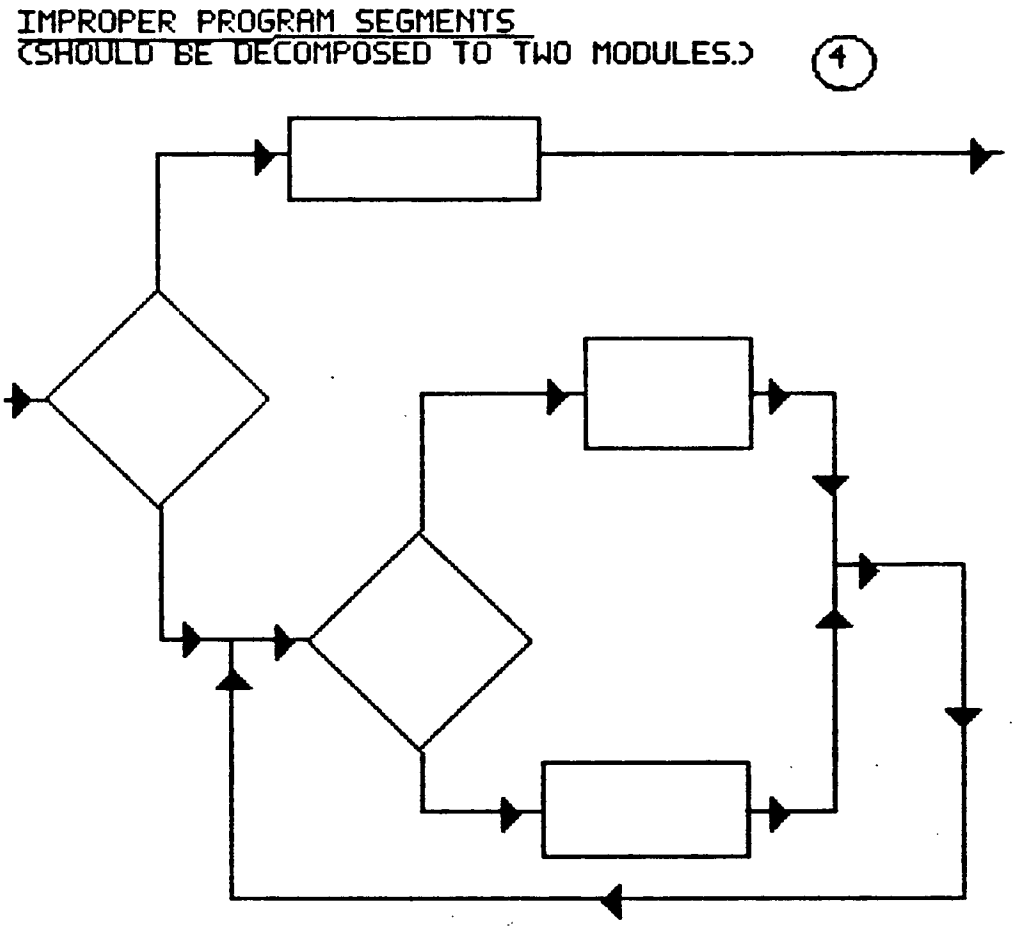


Fig. 1.7 (Cont.) Improper "Integration" of Basic Building Blocks

to a simple, easily understood scheme in which control must enter the module at its entry point, must leave the module from its exit point, and must always be passed back to the invoking module.

3. Module construction is standardized according to traditional modularization rules, and legal program control constructs are restricted concatenation, selection, repetition, and a "well-behaved" branch.

4. Each program variable serves only one program purpose, and the scope of a variable (i.e. the set of modules in which the variable is accessed) is apparent and limited.

5. Error processing follows normal control flow except in the case of unrecoverable errors where normal processing cannot continue.

6. Documentation is required in the source code to introduce each module by explaining its function, its data requirements, and its invocation relationship to other modules in the program.

1.7 STRUCTURED PROGRAMMING CODING STANDARDS

1. The program is divided into independent pieces called modules.

2. A module is a self-contained unit whose code is physically and logically separate from the code of any other module in the program.

   a) A module represents one unique logical program function.

   b) The size of a module should not exceed 100 instructions.

   c) A module is bounded by one entry point and one exit point. During the execution, the program control can enter a module only at its entry point and can leave the module only from its exit point.

3. Modules are related to one another in a hierarchical control structure. Each level in the control structure represents a more detailed functional description of what the program does. It also dictates the transfer of program control from module to module during execution.

Level-1 (top of the hierarchy) contains one and only one program module. Logically, this module represents the overall program structure and contains the "mainline" code for the program. Program execution always begins with this module.

Level-2 contains modules that are performed to execute the overall program function. The modules at level-2 can be executed only by transferring control to them from the mainline module. Execution cannot cause control to "fall" into a module.

Level-3 modules represent functions required to further define the functions at level-2. Control is transferred to level-3 modules only from level-2 modules. This scheme continues from level to level down the entire hierarchical structure.

Program control is always transferred from a module at one level to a module at the next successively lower level (e.g.

from level-3 to level-4). When a module completes executing its code, control is always returned to the module that "called" it.

No loops are allowed in the control structure. This means that a module cannot call itself, nor can it call any module that has called it.

4. Each module should begin with a comment block explaining the function that the module performs, the values passed to the module, the values returned, the modules that called this module, and the modules that this module calls.

5. Comments embedded in the module code should be separated from instructions by one blank line.

6. All comments should be meaningful (e.g. a meaningful comment does not state that this is an add instruction).

7. Avoid unnecessary labels; do not use labels as comments.

8. All variable and module names should be meaningful. Module names should suggest the logical function they perform (e.g. INVERT) and variable names should suggest their purpose in the program (e.g. SHEAR or V).

9. Names of variables that belong to the same table or that are local (i.e. used only in one module) should begin with the same prefix. (Editorial comments: This rule is highly desirable, idealistic, but quite impractical. May clash with #8 above!!!)

10. The only allowable control constructs are concatenation, selection, repetition, and branch. (Editorial comment: This rule, as it is worded, is too rigid and clashes with the current programming practice of the professionals.)

11. At most, one instruction is coded on a line. If an instruction requires more than one line, successive lines are indented.

12. IF-statements should not be nested more than three levels. (Editorial comment: This rule could be considered as a target goal. For some cases it is impractical!!!)

13. The scope of a GOTO statement (branch instruction) should be limited to the module in which it occurs. This means that the GOTO should not be used to transfer control from one module to another; it is used only to branch to the entry point or the exit point off the module in which it occurs.

14. *NONSTANDARD LANGUAGE FEATURES SHOULD NOT BE USED AS A GENERAL RULE. (This is a cardinal rule of programming.)*

15. *OBSCURE (TRICK) CODE SHOULD BE AVOIDED.*

2. SOFTWARE DEBUGGING

The debugging, or "reprogramming(!)," of any given software has negative connotations to the uninitiated, however, this activity is carried out almost perpetually if the "computing environment" is not static. For example, if a given software is not fully independent of the operating system, the change in the operating system may require the recompilation of the program. Even though the program might have been executing satisfactorily prior to this change in operating system, after the change the program may or may not give the correct answers. If the program had been coded robustly and followed all the guidelines discussed in CE 309, then at worst only a recompilation would have been required. However, since most "completed" programs are far from being ideal, prior to the recompilation modifications are usually needed. The changes in the program that need to be undertaken can be labeled as "debugging" under the new environment.

If a program which was executing fully satisfactorily in a given computer configuration has to be moved to another computer system, then changes to make the program run again may be inevitable. If the program was fully portable (an extremely elusive goal to attain) than the source code should not require any modifications prior to recompilation in the new environment. Sometimes changes in the hardware and software environment necessitate changes in the program. If, for example, a program was making use of the plotting devices, an upgrading of the plotting device may require substantial changes in the parts of the program using the plotting devices. Numerous possible permutations on the compiler, operating system, peripheral devices, etc. may lead to the need to "debug" the program.

The "status" of the program may require different approaches in debugging:

(a) You may have been "given," or more likely "leased," only the "load module" of the program. You may not have any access in any form or fashion to the source code. Under these circumstances, you need to contact the "software vendor" for a new load module that will execute under the new computing environment. This action may require "funds" for the transaction, in addition to the regular monthly or annual outlays. Extensive testing (?) of the new version of the program is supposed to have been completed by the software vendor. However, select testing of the new load module of the program by the principal users is in order.

The cost of these tests would be borne by the users. (Example: ANSYS in the Civil Engineering Department's CAE-Laboratory, NASTRAN at LUCC.)

(b) Access to the source code may be restricted. Even though you may be the major user of the package you may not have permission to have access to the source code, even to the listing of the source code. Changes to the source code may be undertaken by the "systems programmers," who may be totally unfamiliar with the "subject area" to which the program is applicable. In a situation like this, when the need for the change to the source code is made and if the program has recompiled, you need to test the new version with as many "case studies" as possible. The cost of these tests should be borne by the systems group and/or the developers. If the software developers are not contractually required to make the changes, then the changes may not be implemented up until the first "opportune period(?)." (Example: Programs FLMXPK, SAP4CNK, SPLT, and ADINA at LUCC.)

(c) If you have full access to the source code, and if you are responsible for the maintenance of the software, the changes in the code will have to be undertaken by you. At the completion of the reprogramming activity, extensive tests need to be conducted by you to confirm the reliability of the results. The cost of these tests may be borne by the computer center only if the computer system change is substantial and during the interim period you are "invited" to test the new configuration. This is a short duration window of opportunity, and the tests can be conducted only at select times and select dates. Other than this possibility, the cost of the tests is usually borne by the software developer.

2.1 COMPILATION AND EXECUTION ERRORS

In order to initiate any debugging you have to have access to the source code. If this is not the case, the only realistic action you can take is to inform the software vendor/developer of the execution error you have encountered, and seek assistance.

If you have access to the source code and if you have to initiate debugging operations, the following paragraphs should be noted, and implemented.

2.1.1 Compilation Errors:

In debugging during the compilation all fatal errors must be eliminated. Various compilers generate informative messages for non-fatal errors. A non-fatal error corresponds to an "operation," statement, usage, etc., that is not fully acceptable

to the compiler, and results in the compiler making an "assumption" in the interpretation of the "code." Even though most non-fatal errors may not have any detrimental effect on the results of your program, all non-fatal errors should be expunged through rewriting the source code, and recompiling the program. After the successful compilation of the program with linkage, and with or without execution, the following checks must be performed:

(a) "Maps" of each module must be carefully examined. The reasons for "unused" variables must be determined. This could be due to "clerical errors," e.g. typing IO instead of IU. Should that be the case, the likelihood of execution error arises. All unused and undefined variables must be taken care of.

(b) Any error messages emanating from "linker" and "loader" must be checked. Corrective actions must be taken to eliminate these messages in the subsequent submissions.

At the completion of the above activities the program is ready to be executed. However, this does not indicate whether the source code is correct or not.

2.1.2 Execution Errors:
Debugging of the program if execution errors are encountered can be trivial, or as in most cases, extremely painful. To isolate the execution errors a logical approach, as itemized later in this document, can and should be employed. The most qualified individuals to identify the source of the execution errors are not the experts at any given computing center, but instead are the individuals who are intimately familiar with (a) the employed solution scheme, (b) the source code in question, and (c) the programming language employed.

2.2 POSSIBLE SOURCES OF ERRORS

The sources of possible errors in the "source code" could be due to almost any predictable or unpredictable reasons and "accidents." However, based on past experience software engineers have developed "classes of errors" or "error categories." Different sources tend to quote different causes and classifications. The list presented below is taken from the book by Martin and McClure. It should be remembered that this is not an all-inclusive list, but is merely a general guideline.

2.3 ERROR CATEGORIES

I. Design Error
   1. Missing cases or steps
   2. Inadequate checking/editing
   3. Initialization error
   4. Loop counter error
   5. Misunderstanding of "specifications"
   6. Incorrect algorithm (e.g. math error)
   7. Timing problems
   8. Failure to consider all data types

II. Coding Error
   1. Misunderstanding of "design"
   2. I/O format error
   3. Control structure error
   4. Syntax error
   5. Incorrect subroutine usage
   6. Initialization/reinitialization errors (e.g. incorrect "flagging")
   7. Indexing/subscripting error
   8. Naming inconsistency
   9. Inadequate checking/editing
   10. Error in parameter passing
   11. Using wrong arithmetic mode
   12. Overflow, underflow, truncation

III. Clerical Error
   1. "Slip" of pencil (misspelling)
   2. Keypunch/data entry

IV. Debugging Error
   1. Insufficient or incorrect use of test cases
   2. Negligence
   3. Misinterpretation of error source/debugging results

V. Testing Error
   1. Inadequate test cases/data
   2. Misinterpretation of test results
   3. Misinterpretation of program specifications
   4. Negligence

VI. External
   1. Hardware failure
   2. Software reaction to hardware failure
   3. Problems in other systems that interfaced with this one

VII. Specification Error
   1. Incomplete or ambiguous specification
   2. Incorrect problem definition

2.4 DEBUGGING GUIDELINES

Almost any and all debugging activities, especially the ones initiated by individuals who do not have a good grasp of software engineering, tend to be disjointed. If the individual is under constant pressure to debug the program to meet deadlines, the efforts may be futile due to the disorganized attack to the bugs. To remedy the situation, below you will find some specific guidelines that must be observed. This list is taken from Martin and McClure's book.

I. Do not use a random approach to debugging. Begin by excluding the unlikely sources of the error. First eliminate the simple cases, and then move on to the more difficult cases.

II. Isolate one error at a time.

III. Employ defensive programming by making program errors easy to locate with the use of debugging code embedded in the program (e.g. printout of selective variable values, logic traces, "end-of-program logic" message). After debugging is completed, leave the debugging code in the program by changing each debugging statement into a nonexecutable comment so they are available for future use but do not interfere with normal processing.

IV. Carefully study actual program output, comparing it to samples of expected output. Many errors are observable in the output listings.

V. Focus attention on data handled by the program rather than solely on program processing logic. Focus on boundary and invalid-input conditions when checking for data-related errors. Check data type, data value ranges, data field size, and data value.

VI. Use the most powerful debugging tools available and a variety of debugging methods (e.g. computer-based and non-computer-based) to avoid becoming locked into considering only one possibility too prematurely.

VII. Keep a record of errors detected and corrected, noting where the errors occurred in the program and the types of errors that were found, since this information can be used to predict where future errors will occur.

VIII. Measure program complexity. Programs (modules) with high complexity have greater propensity for error and will probably require more time to detect and correct errors. Programs (modules) with high complexity are more likely to contain specifications/design type errors, whereas programs (modules) with low complexity are more likely to contain

clerical/coding type errors.

(IX. Use programs artifically seeded with errors to train programmers in debugging techniques and then give them immediate feedback on all seeded errors, showing them what they missed. This is used in "computer center" operations as well as in software engineering courses.)

3. OPTIMIZATION OF THE FORTRAN 77 SOURCE CODE

3.1 WHAT IS OPTIMIZATION

The source code written by the "programmer" needs to be "translated" into a form which will be understandable to the hardware. Compilers will perform this operation. The source code is operated on by the compiler to generate a new code, i.e. program, which is called the "object code," or the object module. In the generation of the object code the compilers can catch some "inefficient" FORTRAN statements or program segments. If the compiler is instructed to do so, the compiler will replace the inefficient code by an efficient code. This is optimization.

The levels of optimization are defined by the extent of "replacements" that will be performed. Even the lowest level optimization, i.e. no optimization, contains limited amount of improvements. For example, a statement like X=((Y+5.-3.))**(1./3.) will be modified to X=(Y+2.)**0.333333333333333. At higher levels of optimization, the code that needs to be modified is not so readily noticeable.

3.2 OPTIMIZING COMPILERS

All FORTRAN compilers running on miniframes and mainframes have optimization capabilities. Some of the microcomputer-based compilers have started to have some optimization capabilities. As the compilers perform higher and higher levels of optimization and develop the corresponding object code, the possibility of "generating" an object code that will not exactly do what was specified by the source code arises.

In CDC NOS (Network Operating System) there are three levels of optimization; the highest, OPT=3, can be used with "caution." The new operating system, VE (Virtual Environment), that is being developed by CDC, and is currently being tested at LUCC, does not have OPT=3. Rather than having an object code of extreme speed, which may not be fully reliable, the approach is in the direction of the generation of the fastest object code that is fully reliable.

3.3 WHY DO WE NEED OPTIMIZATION

It is known that compilation of a program without optimization takes less time than its compilation with optimization. Thus as far as the cost of compilation is concerned, compilation without optimization is a better choice! However, if the object code

generated after the compilation is to be executed it is not possible to make a categorical statement as to which one will be less costly.

If a program is to perform a substantial number of repetitive operations, any optimization of these basic operations will substantially reduce the total computational cost. For example, if a program is coded to solve a "quadratic equation" once, the time savings involved via optimization may not be perceptible, and also may not be necessary.

Assume that a nonlinear finite element analysis of a three-dimensional framed structure is to be conducted. Further assume that there are 1,000 beam-column elements with 6-degrees of freedom per node. The structure is to be analyzed for 50 load steps, and three iterations will be used per load step. Let us also assume that the programming strategy employed is rather routine, and the stiffness matrix for each element and the global stiffness matrix will be reassembled for each iteration cycle of each load step.

If we recall the formula for the computation of the element stiffness matrix, i.e. $[B]^T[D][B]$, we will note that there are two matrix multiplications per element. If "x-nanoseconds" can be shaven of by using optimizaion from each matrix multiplication, than the total savings will be: (x-nanoseconds)*(2 matrix multiplications per element)*(1,000 elements)*(three iteration cycles)*(50 load steps)= 300,000 * "x-nanoseconds." Thus if a set of statements is to be executed too many times, any improvement on its performance will have a noticeable positive effect on the total cost of the job.

For the type of program and problem described above, it will not be unrealistic to say that "unoptimized compilation" will take "1-unit of time," and the execution will require 100-units of time. If optimization is to be used, the optimization time may very well be 1.5-2 units of time. The execution time may very well be 70-units of time. Thus the unoptimized run may cost $101, and the optimized run will be $72.

> ***If the execution of the program is far longer than its compilation, this program is a candidate for some form of optimization.***

3.4 GUIDELINES FOR OPTIMIZATION

The following is a list of guidelines to assist in the decision making process on whether the optimization should be performed or not.

(I) The optimization is performed during the compilation phase. If the compiler is instructed to perform higher

levels of optimization the amount of time required for the compilation, which also includes the optimization, goes up.

(II) During the initial debugging phase of program development the use of optimization may not be prudent. This is especially the case if you are still obtaining compilation error messages.

(III) During the later parts of the debugging, where you may be getting execution errors, the optimization could be used with the following proviso. If the time required for the execution of the program is substantially longer than the time required to compile the program, some level of optimization may be justifiable. If the additional time required for the optimization is less than the execution time, than the optimization is a viable alternative.

(IV) During the final testing of a program with different data sets the decision regarding the different levels of optimization will be similar to the above argument.

(V) When the program is "fully debugged(!!!)" and ready to be installed in object code form, then the program should be optimized to the most "reliable" level.

3.5 LEVELS OF OPTIMIZATION IN CDC NOS FORTRAN77 COMPILER

3.5.1 OPT=0 Compilation

This mode corresponds to "no-optimization," i.e. default value. During compilation constant subexpressions and redundant instructions are removed. These checks are made at the "statement-level." For example, X=((Y+5.-3.))**(1./3.) is modified to X=(Y+2.)**0.333333333333333.

It should be noted that most of the optimization performed at this level could have been done by "careful" coding of the program.

3.5.2 OPT=1 Compilation

In addition to the OPT=0 level optimization, the following are performed.

(1) Redundant instructions and expressions within a sequence of statements are eliminated.

(2) PERT critical path scheduling is done to utilize the multiple functional units efficiently.

(3) Subscript calculations are simplified, and values of simple integer variables are stored in machine registers throughout the loop execution, for innermost loops satisfy-

ing all of the following conditions:

a) Having no entries other than by normal entry at the beginning of the loop.

b) Having no exits other than by normal termination at the end of the loop.

c) Having no external references (user function references or subroutine calls; input/output, STOP, or PAUSE statement; or intrinsic function references) in the loop.

d) Having no IF or GOTO statement in the loop branching backward to a statement appearing previously in the loop.

It should be noted that (a) above could have been done by careful programming. However, (b) is not normally considered by the programmers. Thus, the optimization by the compiler would be far more efficient than the amateur attempts. In order to do (c), you need to program in assembler language, thus optimization by the compiler is a highly desirable alternative.

It can be seen that OPT=0 was done at statement level; whereas, OPT=1 is carried out on program segments.

3.5.3 OPT=3 Compilation

In OPT=3 compilation mode, the compiler performs certain optimizations which are POTENTIALLY UNSAFE. The following optimizations are performed in addition to those provided by OPT=2.

(1) In small loops, indexed array references are prefetched unconditionally WITHOUT ANY SAFETY CHECKS.

(2) When an intrinsic function is referenced, the compiler assumes that the contents of certain "B-registers" are preserved for use following the function processing.

In a loop, the registers available for assignment are determined by presence or absence of external references. External references are user function references and subroutine calls, input/output statements, and intrinsic functions (SIN, COS, SQRT, EXP, etc.).

When OPT=3 is not selected, the compiler assumes that any external reference modifies all registers; therefore, it does not expect any register contents to be preserved across function calls.

If a math library other than FORTRAN Common Library is used in an installation to supply intrinsic functions, the B-Register portion of the OPT=3 option must be deactivated by an installation option in order to ensure correct object code.

PAST EXPERIENCE USING "FULL OPTIMIZATION," I.E. OPT=3 FOR THE CURRENT CONFIGURATION, IN CYBER SYSTEMS GIVES EXCELLENT RESULTS WHEN THE OPTIMIZATION WORKS. THERE ARE CASES WHERE THE OPTIMIZED OBJECT CODE DID NOT CORRESPOND TO THE "FUNCTIONAL REQUIREMENTS" OF THE SOURCE CODE. THUS, IT IS RECOMMENDED THAT OPT=3 SHOULD BE USED WITH GREAT CARE AND CAUTION.

3.6 PROGRAMMING STRATEGIES FOR OPTIMAL SOURCE CODE

The recommendations contained in this section are known to improve the execution time required of the programs written in FORTRAN 77 running under NOS. Some of the recommendations were also tested in select minicomputers, and it was observed that there was a noticeable improvement in the performance.

If the following guidelines are observed in the development of the FORTRAN 77 source code, the program will run faster.

(1) Since the arrays are stored in columnar mode, DO-loops (including implied DO-loops in input/output lists) which manipulate multidimensional arrays should be nested so that the range of the DO-loop indexing over the first subscript is executed first. Implied DO-loop increments should be "one" whenever possible.

Example: Poor practice:

```
      DIMENSION A(20,30,40), B(20,30,40)
      ..................................
      ..................................
      DO 10 I=1,20
      DO 10 J=1,30
      DO 10 K=1,40
   10 A(I,J,K)=B(I,J,K)
```

Good practice:

```
      DIMENSION A(20,30,40), B(20,30,40)
      ..................................
      ..................................
      DO 10 K=1,40
      DO 10 J=1,30
      DO 10 I=1,20
   10 A(I,J,K)=B(I,J,K)
```

(2) The number of different variable names in the subscript expressions should be minimized. For example:

```
X=A(I+1,I-1) + A(I-1,I+1)
```

is more efficient than:

```
IP1=I+1
IM1=I-1
X=A(IP1,IM1)+A(IM1,IP1)
```

NOTE: It is also known that if the subscript expressions are overly complicated, than it may also correspond to an undesirable situation. Caution should be used in the interpretation of the above guideline.

(3) The use of EQUIVALENCE statements should be avoided.

(4) COMMON blocks should not be used as a scratch storage area for simple variables.

NOTE: The software packages that were initially developed in the early seventies, e.g. FLMXPK, had scratch variables for each subroutine in a LABELED COMMON block. This permitted the saving of a dozen or so address locations per subroutine. This approach is redundant in today's programming approach.

(5) Program logic should be kept simple and straightforward.

NOTE: The basic premise of structured top-down programming practice requires that "spaghetti-like logic" shall not be employed. The execution of the program, or the subroutines, should start at the top, and proceed until the end of this program unit; during which time the control should not be transferred to an earlier part in the program. A program that is being written from scratch can follow the guideline. However, most of the earlier vintage programs, especially those that have been "enhanced" at various times, tend to have a spaghetti-like logic.

(6) Program unit, i.e. the main program or a subroutine, length should be less than about 600 executable statements.

NOTE-I: However desirable, the above guideline is sometimes impractical. The Wegmueller-Kostem plate bending finite element has 24 degrees of freedom. The

element stiffness matrix can be populated in one master subroutine. In very crude terms 576 statements, exclusive of any other statement to control the logic, are needed to populate the matrix. Thus there are some "algorithms" that will inevitably need more than 600 executable statements to perform the assigned mission.

NOTE-II: The system-programmers or certain virtual memory minicomputers recommend that after full debugging of the program, the subroutines should be eliminated, and the whole program should be a "continuous string." This type of an argument is highly impractical; however, it indicates that the optimal program for one computer with a given operating system may not, actually will not, be optimal for another configuration. Programs need to be "tuned" for different configurations.

(7) The use of dummy arguments (formal parameters) should be avoided if possible.

NOTE: This guideline somehow conflicts with the suggestions to use variable dimensions, where applicable. The rule can be slightly altered: "Keep the number of formal parameters at a minimum."

(8) The variable dimensions should be avoided if possible; COMMON or local variables should be used instead.

NOTE: The exclusion of variable dimensions will create havoc in programming. This guideline should be discarded for the sake of "convenience."

4. EFFICIENCY OF SOFTWARE

Efficiency is defined as the extent to which a program performs its intended functions without wasting machine resources such as memory, mass storage utilization, channel capacity, and execution time. Efficiency is important but should not be carried to an extreme. Many programmers are unnecessarily concerned with machine efficiency considerations. This obsession with tuning programs to achieve some optimal level of efficiency by playing off time and space requirements is a questionable investment and expense.

The issue above pertaining to dependency, or heavy dependency, on the specific options of the operating system and or compiler needs further elaboration. As will be discussed later in this document, it is highly desirable that the program must be as portable as possible. If the program is heavily dependent on an existing system, then to move this program to another computer facility will be a very painful proposition. Even much worse, for a given computer configuration the vendor will periodically "update" the compiler and the operating system. If each update requires rewriting parts of the program, i.e. debugging, then this program's efficency per run may be good, but over the life of the use of the program it has been an inefficient one because of the continuous investments of time required.

The second issue that requires careful addresing is the possibility of using the OPTIMIZING COMPILER as a crutch for inefficient and sloppy programs. The programmer should try to code the program as efficiently as possible, without being to dependent on the one-of-a-kind features of the operating system and/or compiler. The optimization of this code will enhance its performance.

4.1 EFFICIENCY CHECKLIST

1. Is the program modularized and well-structured?

2. Does the program have a high degree of locality - that is, does the program use only a small subset of its pages at any point during execution - to aid efficient use of virtual memory?

3. Are unused labels and expressions eliminated to take full advantage of compiler optimization?

4. Are exception routines and error-handling routines isolated in separate modules?

5. Was the program compiled with the use of an optimizing compiler?

6. Was as much initialization (e.g., initializing arrays, variables, storage allocations) as possible done at compilation time?

7. Is all invariant code, that is, code which does not need to be processed within a loop, processed outside the loop?

8. Are fast mathematical operations substituted for slower ones? (For example, I+I is faster than 2*I.)

9. Is integer arithmetic instead of floating-point arithmetic used when possible?

10. Are mixed data types in arithmetic or logical operations avoided when possible to eliminate unnecessary conversions?

11. Are decimal points of operands used in arithmetic aligned when possible?

12. Are program variables aligned in storage?

13. Does the program avoid nonstandard subroutine or function calls?

14. In a n-way branch construct, is the most likely condition to be .TRUE. tested first?

15. In a complex logical condition, is the most likely .TRUE. expression tested first?

16. Are the most efficient data types used for subscripts?

17. Are input/output files blocked efficiently?

5. RELIABILITY OF SOFTWARE

Reliability is defined as the extent to which a program correctly performs its functions in a manner intended by the users as interpreted by its designers. Total reliability remains a goal rather than an actuality in virtually all "real-world" software. There currently exist no means for guaranteeing 100% reliability or for measuring exactly how reliable a program is.

5.1 RELIABILITY CHECKLIST

1. Does the program contain checks for potentially undefined arithmetic operations (e.g., division by zero)?

2. Are loop termination and multiple transfer index parameter ranges tested before they are used?

3. Are subscript ranges tested before they are used?

4. Are error recovery and restart procedures included?

5. Are numerical methods sufficiently accurate?

6. Are input data validated?

7. Are test results satisfactory (i.e., do actual output results correspond exactly to expected results)?

8. Do tests show that most execution paths have been exercised during testing?

9. Do tests concentrate on the most complex modules and most complex module interfaces?

10. Do tests cover the normal, extreme, and exceptional processing cases?

11. Was the program tested with real as well as contrived data?

12. Does the program make use of standard library routines rather than develop its own code to perform commonly used functions?

6. UNDERSTANDABILITY OF COMPUTER SOFTWARE

Understandability is defined as the ease with which we can understand the function of a program and how it achieves this function by reading the program source code and its associated documentation.

A simple rule for the understandability of the software is 90-10 rule, which was suggested by Sheiderman (Human Factors in Computer and Information Systems, by B. Sheiderman, Winthrop Publishers, Cambridge, MA, 1980):

> ***A competent programmer should be able to functionally reconstruct from memory 90% of the module after 10 minutes of examining the source code.***

The following checklist can be used to assess the "understandability" of any given source code. Detailed discussions of the entries can be found in Characteristics of Software Quality, by B. Boehm, J. Brown, H. Kaspar, M. Lipow, J. MacLeod and M. Menit, TRW/North Holland Publishing Co., 1978. The greater the number of "yes" answers, the greater the understandability of the software.

6.1 UNDERSTANDABILITY CHECKLIST

6.1.1 Structuredness

1. Is the program modularized and well-structured?

6.1.2 Documentation

2. Is the program documented? Minimal documentation for a well-structured program requires a comment block for each module, subroutine, or subprogram that explains:

   (a) What the module does in one or two brief sentences.

   (b) A list of program variables whose values may be modified in this module.

   (c) A list of modules that invoke this module.

(d) A list of modules that this module invokes.

3. Is other useful commentary material included in this program? This would include:

(a) Inputs and outputs

(b) Accuracy checks

(c) Limitations and restrictions

(d) Assumptions

(e) Error recovery procedures for all foreseeable error exist

(f) Modification history

(g) Date written and date last changed

6.1.3 Consistency

4. Is a consistent indentation and spacing style used throughout the program?

5. Is there at most one executable statement per line of code?

6. Are all variable names and procedure names unique, descriptive, and in compliance with company standards?

7. Does each variable and each procedure have one and only one unique name in the program?

8. Is each variable used to represent one and only one quantity, and is each procedure used to represent one and only one logical function?

9. Is the program a true representation of the design; that is, is the integrity of the design preserved throughout the entire program?

(a) Does the program neither add to nor subtract from the design algorithm?

(b) Is the design structure exactly and explicitly represented in the code?

10. Are all elements of an array/table functionally related?

11. Are parentheses used to clarify the evaluation order of complex arithmetic and logical expressions?

6.1.4 Completeness

12. Are cross-reference listings of variable names and a map of calling and called subroutines supplied?

13. Are all external references resolvable and all input/output descriptions available?

14. Does the program contain all referenced subprograms not available in the usual system library?

15. Are all unusual termination codes described?

16. Are error recovery procedures included?

17. Are error messages descriptive and clearly displayed?

6.1.5 Conciseness

18. Is all code reachable?

19. Are all variables necessary?

20. Is redundant code avoided by creating common modules/subroutines?

21. Is there a transfer to all labels?

22. Is division of the program into an excessive number of modules, overlays, functions, or subroutines avoided?

23. Are expressions factored to avoid unnecessary repetition of common subexpressions?

24. Does the program avoid performing complementary operations on the same variable(s) such that removal of these operations leaves the program unchanged?

25. Does the program avoid poorly understood and nonstandard language features?

6.2 COMMENTS ON THE MODULE SIZE

There have been a number of suggestions regarding how big a module (sometimes translated as "subprogram") is "tolerable":

(a) The old rule of thumb was that a routine should not be larger than the number of cards that can be gripped in one hand between the thumb and the index-finger. (Highly unscientific and depends on your "anatomy.")

(b) CDC: Should not exceed 100 lines (FORTRAN77 Reference Manual).

(c) IBM: Should not exceed 50 lines (B. Boehm, "Seven Basic Principles of Software Engineering," in Infotech State-of-the-Art Reports: Software Engineering Techniques, Infotech International, Maidenhead, England, 1977).

(d) Practical suggestions: The length of the source code should not exceed one page of printed lines.

(e) A counter-observation: Wegmuller-Kostem plate bending finite element has 24-degrees of freedom. The terms of the element stiffness matrix can be defined in one subprogram. Just the listing of elements requires 576 lines. Additional lines of the code are essential for dimensioning, populating the elements, etc. If this subroutine is broken into a smaller subroutine, it will adversely affect its "readability." Thus there are some operations that should not be broken down merely for the size of the module.

7. MODIFIABILITY OF SOFTWARE

Modifiability is defined as the ease with which a program can be changed. A programmer has a low probability of success when modifying a program. If the modification involves fewer than 10 program instructions, the probability of correctly changing the program in the first attempt is approximately 50%; but if the modifications involve 50 program instructions, the probability drops to 20% (J. Munson, "Software Maintainability: A Practical Concern for Life-Cycle Costs," Proceedings of IEEE 2nd International Computer Software and Application Conference, Chicago, 1978).

5.1 MODIFIABILITY CHECKLIST

1. Is the program modular and well-structured?

2. Is the program understandable?

3. Does the program avoid using literal constants in arithmetic expressions, logical expressions, size of tables/arrays, and input/output device designators?

4. Is there additional memory capacity available to support program extensions?

5. Is information provided to evaluate the impact of change and to identify which portions of the program must be modified to accommodate the change?

6. Is redundant code avoided by creating common modules/subroutines?

7. Does the program use standard library routines to provide commonly used functions?

8. Does the program possess the quality of generality in terms of its ability to:

   (a) Execute on different hardware configurations?

   (b) Operate on different input/output formats?

   (c) Function in subset mode performing a selected set of features?

(d) Operate with different data structures or algorithms depending on resource availability?

9. Does the program possess the quality of flexibility in terms of its ability to:

(a) Isolate specialized functions that are likely to change in separate modules?

(b) Provide module interfaces that are insensitive to expected changes in individual functions?

(c) Identify a subset of the system that can be made operational as a part of contingency planning for a smaller computer?

(d) Permit each module function to perform one unique function?

(e) Define module intercommunication based on the function the modules perform, not upon how the modules work internally?

10. Is the use of each variable localized as much as possible?

8. PORTABILITY OF SOFTWARE

Portability is defined as the extent to which a program can be easily and effectively operated in a variety of computing environments.

The criticality of this issue can not be overstated. For example, a program may be developed, or an existing program may be modified, within the scope of a research project. These activities may be carried out in CYBER 730. The program may then have to be "installed" in a virtual 32-bit machine. Or, conversely, a program may have to migrate from a virtual 32-bit machine to CYBER. Major modifications may have to be undertaken. In the case of a major program, say about 50K executable statements, in FORTRAN-IV from CYBER environment to IBM-3084 FORTRAN VM corresponds to many man-months of effort, if not in excess of a man-year effort. CYBER has a relatively straightforward job control language (JCL), where very little, if any, references to the source code and files need to be made at JCL. IBM 3084 FORTRAN VM has a highly complex JCL environment, where extensive referencing to the source code files has to be made.

Regarding the portability, the following "experience" should be remembered. A finite element code for DG MV/10000 with AOS/VS operating system was brought in. The developers indicated there was no DG-AOS/VS version of this program, however, an IBM 4341 version did exist. Both the DG and IBM versions were based on 32-bit virtual configuration, with very similar FORTRAN77 compilers. It was suggested that with slight changes the IBM version should run on DG. In the effort to compile the program, the DG compiler generated fatal error messages about three times the length of the source code. The project was abandoned!

8.1 PORTABILITY CHECKLIST

1. Is the program written in high-level, machine-independent language?

2. Is the program written in a widely used standardized programming language, and does the program use only a standard version and features of that language?

3. Does the program use only standard, universally available library functions and subroutines?

4. Does the program use operating system functions minimally or not at all?

5. Are program computations independent of word size for achievement of required precision or memory-size restrictions?

6. Does the program initialize memory prior to execution?

7. Does the program position input/output devices prior to execution?

8. Does the program isolate and document machine-dependent statements?

9. Is the program structured to allow phased (overlay) operation on a smaller computer?

10. Has dependency on internal bit representation of alphanumeric or special characters been avoided or documented in the program?

8.2 GENERAL COMMENTS

The literature on conversion of software from FORTRAN-IV to FORTRAN77 is limited. The publications by hardware and software vendors tend to lack specificity. One of the few "general" books that can be referred to is by Ingemar Dahlstrand, Software Portability and Standards, John Wiley and Sons, 1984.

Canned automated conversion programs tend to follow Kostem's 95-5 rule; that is, the program will convert 95% of FORTRAN-IV statements that are unacceptable in FORTRAN77. The remaining 5% conversion, which will have to be done "manually," would require 95% of the total conversion effort.

9. TESTABILITY OF SOFTWARE

Testability is defined as the ease with which program correctness can be demonstrated. Thoroughness of testing depends on a careful selection of test cases and is guided by the following rules:

1. Every program instruction and every path should be executed at least once.

2. The more heavily used parts of the program should be tested more thoroughly.

3. All modules should be tested individually before they are combined. Then the paths and intersections between the modules should be tested.

4. Testing should proceed from the simplest to the most complex test cases; that is, tests involving fewer loops and conditions should be performed before tests involving more complicated control constructs and more decisions.

5. Testing of program should include normal processing cases, extremes, and exceptions.

9.1 TESTABILITY CHECKLIST

1. Is the program modularized and well-structured?

2. Is the program understandable?

3. Is the program reliable?

4. Can the program display optional intermediate results?

5. Is program output identified in a clear, descriptive manner?

6. Can the program display all inputs upon request?

7. Does the program contain capability for tracing and displaying logical flow of control?

8. Does the program contain a checkpoint-restart capability?

9. Does the program provide for display of descriptive error messages?

10. USABILITY OF SOFTWARE

Regardless of the high degree of accuracy, efficiency, ease of maintenance, etc. of any given software package, the single most critical attribute of the software is its "usability." If, in order to use the software, the user has to master too many subject areas or be cognizant of many rules regarding the use of the program, than regardless of the potential of the software its use may either be limited, or if the users are required to use it, then the efficient use of the program can not be attained.

In the extreme, it is stated that any given software should not have a users manual(!!!). The program should prompt the user for input data stream, and for the interpretation of the results. For example, the manuals of the finite element program ANSYS take a number of three ring binders. The program also contains an on-line "manual" capability. However, due to the abbreviated nature of the on-line manuals, the complexity of the subject matter dictates the use of the hard-copy manuals. The on-line manual capability is employed by the user during the "session" to check the key points and salient details only. Thus, so far for complicated programs, e.g. nonlinear finite element analysis, "manual-less" programs, espoused by the computer scientists and artificial intelligence experts, have not materialized as yet.

The "cost" associated with the usability of the software may be the single most important factor in the use of the computer in any given process. In the Lientz-Swanson survey of over 500 data-processing departments, almost 50% of all software maintenance work was attributed to user requests, while less than 20% was attributed to software errors (Software Maintenance Management, by B. Lientz and E. Swanson, Addison-Wesley Publishing, Reading, MA, 1980).

10.1 USABILITY CHECKLIST

(Based on "User-Perceived Quality of Interactive Systems," by W. Dzida et al., Proceedings of 3rd International Conference on Software Engineering, May 1975).

1. Is the program self-descriptive from the user perspective?

   (a) Are the explanations of how the program works and what the program does available in different levels of

detail with examples included?

(b) Is the HELP feature pertinent to any dialogue situation included?

(c) Is a correct, complete explanation of each command and/or operating mode available on request?

(d) Can the user become thoroughly acquainted with the program usage without human assistance?

(e) Is current program status information readily available on request?

2. Does the program provide the user with a satisfying and appropriate degree of control over processing?

(a) Does the program permit interruptions of a task to start or resume another task when operating in interactive mode?

(b) Does the program permit process canceling without detrimental or unexpected side effects?

(c) Does the program allow the user to make background processes visible?

(d) Does the program have a command language that is easy to understand and allows clustering of commands to build "macros?"

(e) Does the program provide detailed prompting when requested to help the user find his way through the system?

(f) Does the program provide understandable, non-threatening error messages?

3. Is the program easy to learn to use?

(a) Is the program usable without special data processing knowledge?

(b) Are input formats, requirements, and restrictions completely and clearly explained?

(c) Is user input supported by a menu technique in interactive systems?

(d) Does the program offer error messages with correction hints?

(e) For interactive systems, are manuals "on-line?" For batch systems, are manuals readily available?

(f) Are manuals written using user terminology?

4. Does the program make use of a data management system that automatically performs clerical/housekeeping activities and manages formatting, addressing, and memory organization?

5. Does the program behave consistently in a manner that corresponds to user expectations?

(a) Does the program have a syntactically homogeneous language and error message format?

(b) Does the program behave similarly in similar situations by minimizing variances in response times?

6. Is the program fault-tolerant?

(a) Can the program tolerate typical typing errors?

(b) Can the program accept reduced input when actions are to be repeated?

(c) Can command be abbreviated?

(d) Does the program validate input data?

7. Is the program flexible?

(a) Does the program allow for freeform input?

(b) Does the program provide for repeated use without the need for redundant specifications of input values?

(c) Are variety of output options available to the user?

(d) Does the program provide for omission of unnecessary inputs, computations, and output for optional modes of operation?

(e) Does the program allow the user to extend the command language?

(f) Is the program portable?

(g) Does the program allow the user to define his own set of functions and features?

(h) Can the program be seen in a subset mode?

(i) Does the program allow the experienced user to work with a faster version, allowing abbreviated commands, default values, and so on, and inexperienced users to work with a slower version, providing a help command, monitoring capabilities, and so on?

11. OVERLAYING

11.1 NEED FOR OVERLAYING

The physical size of the central memory of any given computer is not unlimited. As shown in Fig. 11.1, part, if not most, of the physical space of the central core will be taken up by the operating system, and many other resident "routines and files." The indicated available space is what the program, including the DIMENSIONED ARRAYS, can utilize. If the total central core requirement of the program is less than the available space, than the program can compile, link, load and execute without any complication. It should be noted that if all variables are in DOUBLE PRECISION, than the central core requirement for the variables will be twice the amount of the SINGLE PRECISION case. This does not mean that the core requirements will double; only the space needed by the variables will increase.

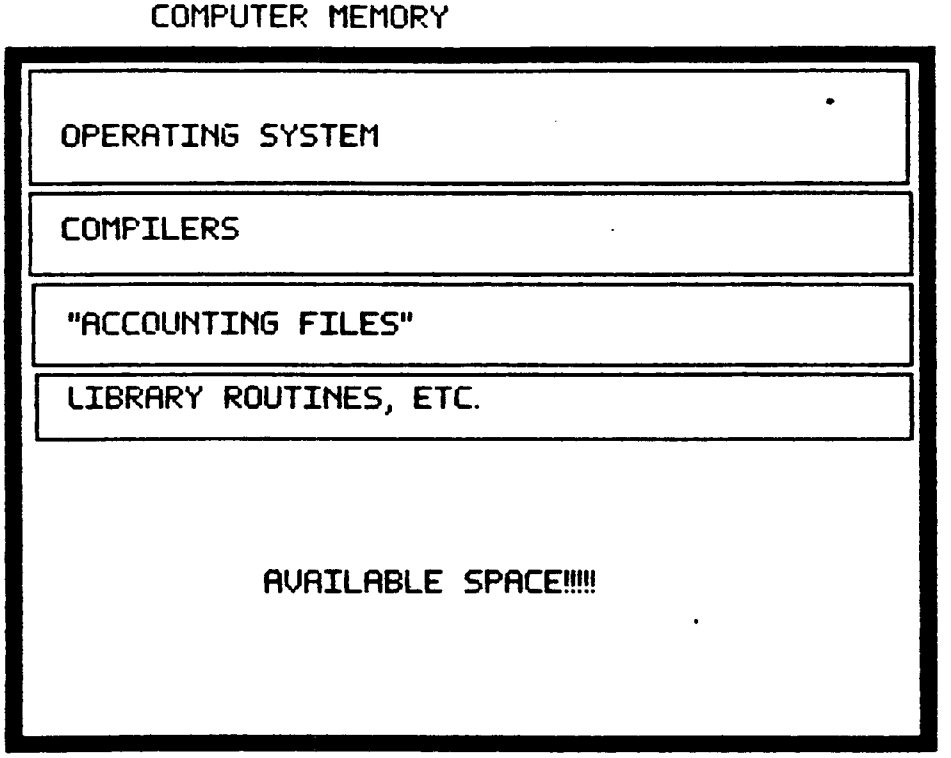


Fig. 11.1 The Use of Non-Virtual Computer Core

If the available central core space is less than the program requirement (see Fig. 11.2), and if the computer system in question is not a virtual system than the program can not be handled by this computer without some modifications. Some of the "scientific computers," e.g. CYBER 850, and most of the personal computers do not have virtual memory. If these computers are to be used for very long programs and/or programs that use many large arrays, such that they will not fit into non-virtual systems, the approach to take will be the use of OVERLAYING. This is a definite need for the problem shown in Fig. 11.2.

The need for overlaying can be illustrated via a past experience with program development in CDC 6400 computer (predecessor of CYBER 850 at the LUCC). In the early seventies this particular computer had 40,960 60-bit words were available to the users. The developed finite element program, even in its pre-production version, required about 230,000 words of central core storage! Due to the long word-length of the computer, there was no need for any DOUBLE PRECISION variables. Due to the programming strategy involved it was possible to develop and execute the program within 35,000 word central storage. Because of the "improvements(!)" to the operating system by the vendor, the program later required about 40,500 words.

> ***In the development of overlayed programs for non-virtual computer systems, the maximum central core required should be kept "sufficiently" below the limitations of the computer. Any future limited changes to the program and/or changes by the vendor to the computer system should not require the reorganization of the program.***

11.2 OVERLAYING CONCEPTS

In order to "fit" the program to the core it is essential that the program on hand be subjected to a major "re-arrangement." This effort is usually quite demanding if the program was developed without any consideration for future overlaying. However, if the program development is conducted with full consideration for overlaying, than this program can be overlayed with great ease.

Furthermore, an overlayed program can be stripped of its overlay features and executed as a "routine" program if needed. Such a need can arise if the program migrates from non-virtual memory computer to a virtual memory computer, e.g. from CYBER 850 to Data General MV/10000.

11.2.1 MAIN and PRIMARY OVERLAYS

In order to overlay a program, first a "main," "core," or "executive" program needs to be identified. This "portion" of

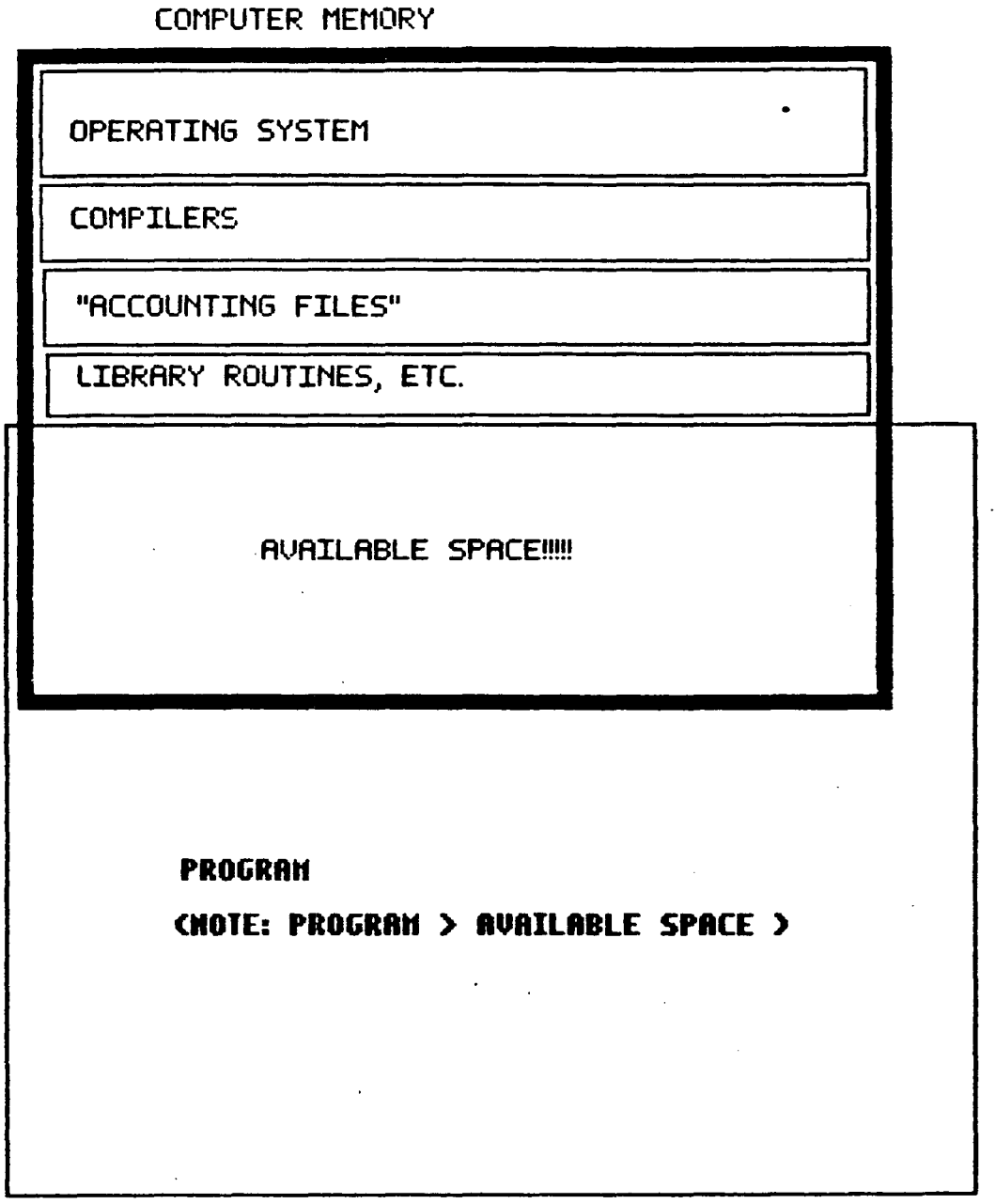


Fig. 11.2 Central Core Requirement of A Large Program

the original program will reside in core for the full duration of the job. This program can be referred to as (OVERLAY 0,0), and is also known as the MAIN OVERLAY. These discussions can best be visualized through inspection of Fig. 11.3. It is then possible to identify major program segments that can reside in the core one at a time in addition to the executive program. OVERLAY-1, OVERLAY-2, etc. can be considered as independent programs, or "mega-subroutines." OVERLAY-1, OVERLAY-2, etc. can be called only from OVERLAY-0, i.e. the executive program. When the execution of OVERLAY-0 reaches such a point that a call is made to OVERLAY-1, OVERLAY-0 and OVERLAY-1 will be in the central core. When this call is made the control of the flow of the program will be transferred to OVERLAY-1. This is similar to the calls that are made to SUBROUTINES. At the completion of the execution of OVERLAY-1 the control will be transferred back to OVERLAY-0. At this stage OVERLAY-1 is "automatically removed" from the central core. The execution of OVERLAY-0 will continue, and a call to OVERLAY-4 may be encountered. OVERLAY-4 will be brought into the core; thus, OVERLAY-0 and OVERLAY-4 will be in core. At the completion of the execution of OVERLAY-4, the control will be transferred back to OVERLAY-0, etc.

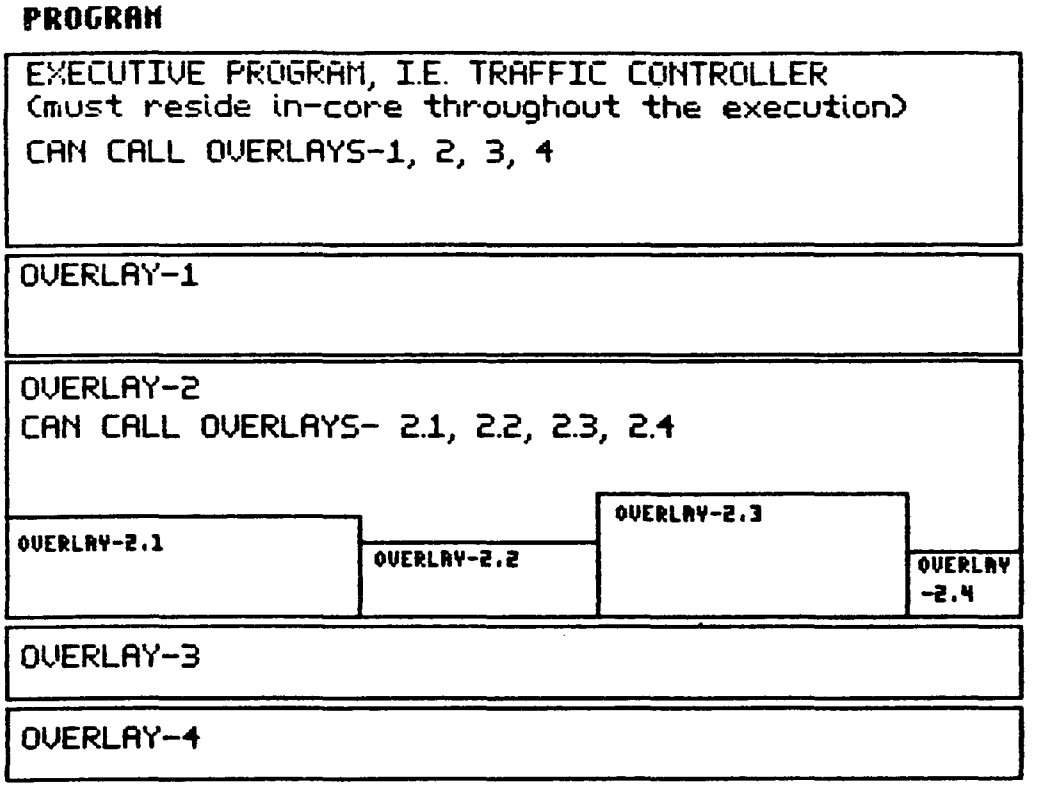


Fig. 11.3 Overlay Structuring of A Large Program

11.2.1 SECONDARY OVERLAYS

It is also possible to identify program segments within a primary overlay that can be considered as "self-contained." Calls to these SECONDARY OVERLAYS can be made from the PRIMARY OVERLAY to which they "belong." The general concept can be seen in Fig. 11.3. It should be noted that, for example, during the execution of OVERLAY-2,3, the overlays that will be in the core are OVERLAY-0, OVERLAY-2, and OVERLAY-2,3.

There are no tertiary overlays.

11.3 PROGRAMMING STATEMENTS

OVERLAY is not an ANSI (American National Standards Institute) approved feature. The material presented in this section is based on FORTRAN 5 (enhanced version of ANSI FORTRAN 77) of Control Data Corporation. The following is a typical program, whose length can be from a few hundred to many hundreds of thousands of lines long:

```
OVERLAY(CE309,0,0)
PROGRAM XXX
.....
.....
CALL OVERLAY(5HCE309,1,0)
.....
.....
CALL OVERLAY(5HCE309,4,0)
.....
....
CALL OVERLAY(5HCE309,2,0)
.....
.....
CALL OVERLAY(5HCE309,3,0)
.....
.....
CALL OVERLAY(5HCE309,1,0)
.....
.....
STOP
END
SUBROUTINE PDQ(.....)
......
......
RETURN
END
[additional subroutines, where needed]
OVERLAY(CE309,1,0)
PROGRAM YYY
....
RETURN
END
```

```
      SUBROUTINE ABC(.....)
      ....
      ....
      RETURN
      END
      [additional subroutines, where needed]
      OVERLAY(CE309,2,0)
      PROGRAM ZZZ
      .....
      .....
      .....
      CALL OVERLAY(5HCE309,2,1)
      .....
      .....
      CALL OVERLAY(5HCE309,2,4)
      .....
      .....
      CALL OVERLAY(5HCE309,2,2)
      .....
      .....
      CALL OVERLAY(5HCE309,2,3)
      PROGRAM YYY
      ....
      ....
      CALL OVERLAY(5HCE309,2,1)
      ....
      ....
      RETURN
      END
      [as many subroutines as needed]
      OVERLAY(5HCE309,3,0)
      PROGRAM WWW
      .....
      .....
      RETURN
      END
      [subroutines, if needed]
      OVERLAY(5HCE309,4,0)
      PROGRAM XYZ
      ......
      ......
      RETURN
      END
```

In the above example CE309 is the name of the file on which generated overlays are to be written. "0,0", "1,0", "2,0", "2,1", "2,2", etc. are the primary and secondary overlay numbers. These numbers are in octal (0 through 77).

There are additional parameters, or switches, on the overlay statement. They are not used frequently enough to justify their presentation herein.

11.4 OVERLAY COMMUNICATIONS

Inspection of Fig. 11.3 and the "program" given in the previous section indicates that there exists a need for a mechanism to transfer the values of the variables from one overlay to another. If such a mechanism does not exist the values read or computed, for example, in OVERLAY(CE309,0,0) can not be transferred to OVERLAY(CE309,1,0).

Communication amongst the overlays, in terms of the transfer of data, can be accomplished in the following two ways:

(a) LABELED COMMON and/or blank COMMON, and/or

(b) files.

In the use of COMMON the values can be assigned to the variables listed in the COMMON in one overlay. These LABELED COMMON and/or blank COMMON must also be listed in the MAIN OVERLAY, i.e. OVERLAY(...,0,0). The overlay which needs this information can have the appropriate COMMON blocks listed. Thus, for example, the values generated in OVERLAY(...,1,0) will, in a sense, be available in the corresponding COMMON blocks in OVERLAY(...,0,0), and when OVERLAY(...,4,0) is activated the data will readily be available if the COMMON blocks are listed in this overlay.

The second approach, which is especially used for large matrices, employs the unformatted file approach. At OVERLAY(...,0,0) necessary files, e.g. TAPE11=11 in FORTRAN 5 terminology, can be defined. In an appropriate overlay the data can be written to this file. In another overlay the file should be rewound (i.e. issue REWIND command) and the data can be read. Since the "file number" is another variable that must be transmitted, it needs to be defined in a LABELED COMMON.

11.5 VIRTUAL COMPUTER SYSTEMS

In virtual configurations the physical size of the central core is not a major limitation if the program has core requirements that are far larger than the core. Part of the "hard disk" is accessible by the program as if it is part of the core. This access is automatically handled by the software system of the computer, and it is totally transparent to the user. Thus, there is no need to overlay the programs. Actually, most new systems do not have any overlay features. All minicomputer systems have virtual memory features. Most advanced and powerful personal computers are in the process of introducing virtual system concepts. On the other hand, MS.FORTRAN, which is available for personal computers with MS.DOS and PC.DOS, has the overlay features. For any scientific and engineering computations the computer system should either have overlay features or should have virtual memory.

11.5.1 Computer Resource Requirements

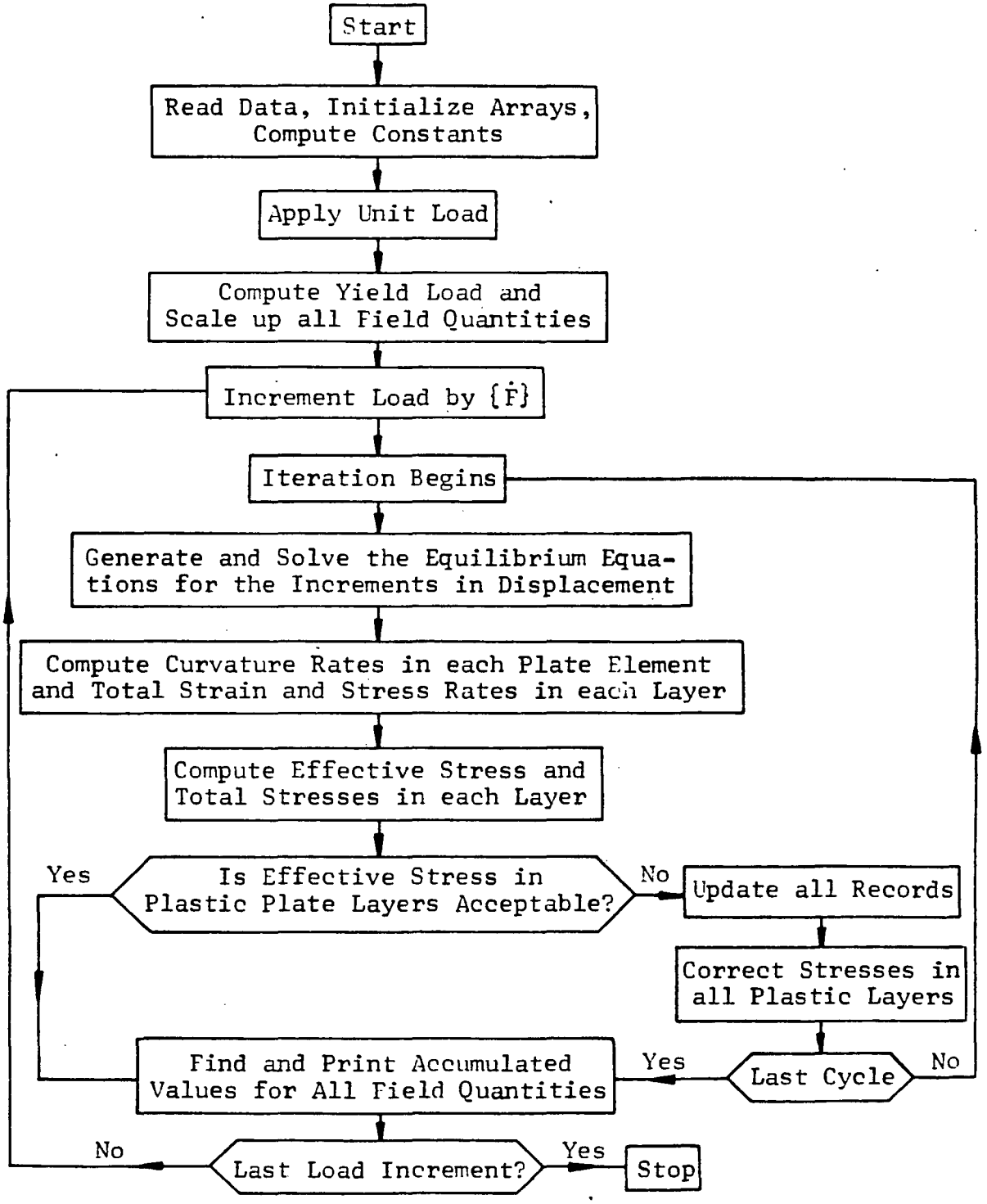
Limited benchmark tests conducted by the author may give some "feel" for the CPU time requirements for overlayed vs. virtual systems. Two finite element programs were fine-tuned for non-virtual memory system, i.e. overlaying was used. A series of finite element problems were executed. The same problems were run using two different virtual memory minicomputers using two different finite element programs. The programs were fine tuned by the developers for these minicomputers. The same problems were executed in these new programs with modified element and/or node point numbering in order to give the best performance as far as the specific program is concerned. The total CPU time for these runs were at least five times larger than the corresponding figures obtained in the non-virtual computer. For some problems the factor was larger than "10." It is recognized that the mini-computers are slower than the scientific mainframes; however, such a discrepancy in the time requirements was unexpected (!!!).

11.5.2 Future Trends

It should be recognized that the trend is towards virtual systems for "decent" personal computers, workstations, minicomputers, and mainframes. It is not unrealistic to expect that in the "near future" there will not be any need for overlaying techniques.

11.6 EXAMPLE PROGRAM

The logical flowchart of a finite element program for the elastic-plastic analysis of eccentrically stiffened plates is given in Fig. 11.4. In this chart almost each "box" corresponds to PRIMARY or SECONDARY OVERLAYs. The first two boxes correspond to the MAIN OVERLAY.


Start
Read Data, Initialize Arrays, Compute Constants
Apply Unit Load
Compute Yield Load and Scale up all Field Quantities
Increment Load by {Ḟ}
Iteration Begins
Generate and Solve the Equilibrium Equations for the Increments in Displacement
Compute Curvature Rates in each Plate Element and Total Strain and Stress Rates in each Layer
Compute Effective Stress and Total Stresses in each Layer
Yes
Is Effective Stress in Plastic Plate Layers Acceptable?
No
Update all Records
Correct Stresses in all Plastic Layers
Last Cycle
Yes
No
Find and Print Accumulated Values for All Field Quantities
No
Last Load Increment?
Yes
Stop