HEDL-SA-2244-FP

MASTER

CONF-810415--9

APPLICATION OF SOFTWARE ENGINEERING TO DEVELOPMENT OF REACTOR-SAFETY CODES

N. P. Wilburn and L. G. Niccoli

November 1980

ANS/ENS International Topical Meeting on Advances in Mathematical Methods for Nuclear Engineering Problems

April 1981, Munich, Federal Republic of Germany

HANFORD ENGINEERING DEVELOPMENT LABORATORY
Operated by Westinghouse Hanford Company, a subsidiary of Westinghouse Electric Corporation, under the Department of Energy Contract No. EY-76-C-14-2170

DISCLAIMER

APPLICATION OF SOFTWARE ENGINEERING TO DEVELOPMENT OF REACTOR-SAFETY CODES

N. P. Wilburn and L. G. Niccoli

Hanford Engineering Development Laboratory

For presentation at the ANS/ENS International Topical Meeting on Advances in Mathematical Methods for Nuclear Engineering Problems, April 27-29, 1981, Munich, Federal Republic of Germany

I. INTRODUCTION

The techniques of Software Engineering are an answer to the drastic increases in programming costs being experienced in both the business and scientific communities. The plot given in Figure 1 shows this cost trend. It was taken from a paper by B. W. Boehm,[1] which was published in 1976. The plot is from 1950 to 1985 and shows, as a percentage of overall computing costs, the cost of programming in the U.S. It reflects first that the cost of software is growing very rapidly, partly because of widespread decreases in hardware costs, but second, because it reflects increased per unit cost of software due to greater complexity of both scientific and business oriented software.

Within the trend to increasing software costs, we are also seeing maintenance cost become increasingly important. (See Figure 2.) In 1955, software development costs were about three-quarters of the total cost of a system, whereas maintenance was only about one-quarter. (Maintenance in this context can be defined first as any changes to a code required in order to make it run as originally specified; and, secondly, as any change made later to the code to make it somehow more efficient or to add extra features which had not originally been planned.) Total cost here is defined as that which is incurred from the inception of the software, until the time the software can be disposed-of or replaced.

Maintenance since 1955 has been a steadily increasing fraction of the total system cost. In 1976 maintenance was approaching half the total cost, and now (1980) it is approximately two-thirds of the total cost. Thus, the biggest part of the increased total cost of software is due to the increasing maintenance cost.

Conversely, the per unit cost of hardware has been decreasing almost exponentially. This is due to the advent of integrated circuit technology and due to the application of engineering technology, in general, to hardware. It can be said that hardware costs have been decreasing because the engineering methodology has been applied, whereas, the cost of software has been increasing because of the lack of the application of engineering. Thus, the current methodology of software development must be severely questioned. First, analysis techniques as applied to scientific software have been rather limited, particularly when compared with the widespread use of analytical methods for hardware design. Second, a modularity discipline, such as is customarily imposed during preliminary design of hardware, is not imposed during preliminary software design. Third, coding of the modules has been generally very unclear and not at all easy to understand. Planned, methodical, thorough testing has been simply the exception in the case of scientific software development.

As a result of the drastically increasing cost of software and the lack of an engineering approach, the technology of Software Engineering is being developed. Software Engineering provides an answer to the increasing cost of developing and maintaining software. It has been applied extensively in the business[2] and aerospace communities[3] and is just now being applied to the development of scientific software and, in particular, to the development of reactor safety codes at HEDL.

II. DESCRIPTION OF SOFTWARE ENGINEERING

The technology of Software Engineering is actually a combination of several disciplines.[4] Primarily, it is a methodology by which a large scale software development project is partitioned into manageable pieces. Secondly, in the course of using these disciplines, the "Top Down" approach is used in all facets. By this approach, major requirements of the software development project are specified first, then details are developed by stepwise refinement. Debugging can start at a high level, rather than at the time of code implementation and testing. Associated with each of these disciplines is a peer review at each stage, which is called a "walkthrough."[5] The walkthrough allows peers to dig into specifications and code at each and every stage of the development and to ferret out bugs at relatively high problem oriented levels.

Software engineering disciplines can be broken down into five categories, which are Survey, Analysis, Design, Implementation, and Testing Phases.

Survey Phase

The survey phase in the development of scientific software involves Algorithm Specification. This would be where the defining model equations were derived for a new code. In the case of a restructuring or re-engineering of an existing code, that is where structured programming constructs would be implemented. The survey also includes activities needed to define a charter for the software development project.

Analysis Phase

The Analysis Phase consists of determining the software specification. This is done in terms of Data-Flow-Diagrams (an example of which is shown in Figure 3); a Data-Dictionary, which defines all of the data flows that are presented on the Data-Flow-Diagrams; and a set of Mini-Specs., which define the processes occurring in the primitive bubbles which appear

on the lowest level of Data-Flow-Diagrams. The methodology of the Analysis Phase has been well established by DeMarco,[6] Yourdon[4] and others and will not be described further in this paper.

Design Phase

The third phase is the Design Phase, where the hierarchial subroutine and executive call lists are derived.[7, 8, 9] An example of hierarchy charts is shown in Figure 4. This takes the familiar form that has been so heavily used in the past. The Design Phase results in modular subroutines with completely specified composition. Traditionally, the latter has not been accomplished. The resulting modules are coherent, small (the order of a page or two of Fortran coding) and as independent as possible. The size of interfaces between the modules are minimized. While a large number of small routines is consistent with small interfaces, it brings a large number of interfaces into being. In contrast, the traditional problem is a small number of large, obscure interfaces. The new situation is much easier to deal with locally, but requires a design methodology to efficiently optimize the more varied possibilities and keep error rates down. Finally, the module coding which ultimately results is thoroughly readable and understandable.

Implementation Phase

The fourth discipline is that of Implementation of the code. This is done in a "Top-Down" fashion. That is, the high level modules on the hierarchy chart are implemented first so that a total system is operational, in a rudimentary fashion, right at the outset. This allows the high level interfaces to be exercised first. It has been shown that most of the bugs that occur in any given piece of software will occur at the interfaces and the most difficult ones to fix will occur at the high level interfaces.[10] Finally, in the course of implementation, standard constructs are used repetitiously.[11] The four basic constructs are shown in Figure 5, and have been shown mathematically to be all that is necessary to create any given computer program.

on the lowest level of Data-Flow-Diagrams. The methodology of the Analysis Phase has been well established by DeMarco,[6] Yourdon[4] and others and will not be described further in this paper.

Design Phase

The third phase is the Design Phase, where the hierarchial subroutine and executive call lists are derived.[7, 8, 9] An example of hierarchy charts is shown in Figure 4. This takes the familiar form that has been so heavily used in the past. The Design Phase results in modular subroutines with completely specified composition. Traditionally, the latter has not been accomplished. The resulting modules are coherent, small (the order of a page or two of Fortran coding) and as independent as possible. The size of interfaces between the modules are minimized. While a large number of small routines is consistent with small interfaces, it brings a large number of interfaces into being. In contrast, the traditional problem is a small number of large, obscure interfaces. The new situation is much easier to deal with locally, but requires a design methodology to efficiently optimize the more varied possibilities and keep error rates down. Finally, the module coding which ultimately results is thoroughly readable and understandable.

Implementation Phase

The fourth discipline is that of Implementation of the code. This is done in a "Top-Down" fashion. That is, the high level modules on the hierarchy chart are implemented first so that a total system is operational, in a rudimentary fashion, right at the outset. This allows the high level interfaces to be exercised first. It has been shown that most of the bugs that occur in any given piece of software will occur at the interfaces and the most difficult ones to fix will occur at the high level interfaces.[10] Finally, in the course of implementation, standard constructs are used repetitiously.[11] The four basic constructs are shown in Figure 5, and have been shown mathematically to be all that is necessary to create any given computer program.

Testing

The fifth discipline is that of Testing. Testing must be done in a systematic fashion.[8] The individual tests must be documented and retrievable. The testing must be designed in order to make the code fail. It should not be done by the developer. The developer naturally wants to show the successful operation of the code, whereas a test must be designed to find the code's weaknesses and conditions where it will fail.

Associated with all of the above disciplines is the Walkthrough.[5] In each one of the five disciplines, the Walkthrough is an integral part which allows peer review of the development of the code at that stage and also allows high level debugging. The Walkthroughs will eliminate most of the major errors in the code.

III. APPLICATIONS OF SOFTWARE ENGINEERING

The techniques of Software Engineering have been applied at the Hanford Engineering Development Laboratory (HEDL) to the development of LMFBR safety codes. These were an existing code--MELT,[12] a new module FUMO-E, and a new very large integrated code system called CONACS. MELT (about 10,000 lines of code) was restructured using the Software Engineering Techniques in order to reduce its maintenance (which had become substantial); to make subroutines more independent and coherent; and, most importantly, to allow for the easy addition of new modules as they were developed to model various physical processes occurring during a hypothetical core disruptive accident (HCDA). The new module FUMO-E ($\sim$3,000 lines of code) was developed to model molten fuel motion occurring internally to a fuel pin during a hypothetical transient overpower accident. The new large system, CONACS (estimated to be $\sim$ 50,000 lines of code) is being developed to model the behavior of containment system compartments under HCDA conditions. This section describes the utilization of Software Engineering methodology as applied to these three codes. The phases introducted in the previous section are used.

During the Algorithm Definition Phase (Survey) the model equations for the processes were derived and the numerical methodology (or Solution Method) established for the new codes FUMO-E and CONACS. In the case of MELT, the existing code defined the algorithm to be used. Operations were defined and complex sequence control structures to the major routines were restated for clarity in terms of the constructs of structured programming. This task consumed approximately one-third of the total project effort.

In the Analysis Phase, the new Data-Flow-Diagrams, the Data Dictionary and Structured English were established for all three codes. These latter components go far beyond the traditional flow charts, which only show sequence. They are much more understandable and contain a far greater amount of information. This is because the Data Flow Diagram is invariant with respect to the program design whereas the flow chart is not. The Data-Flow-Diagram bubbles (or processes) represent mathematical transforms or functions. Flow charts show only sequences of operations, which can be arbitrary, and do not show the underlying transformations. In the case of a large scientific code (with which we are dealing here,) graphical representations as seen in the Data-Flow-Diagrams, are absolutely necessary to display the relationships (functional independence). Data-Flow-Diagrams are much more representative of the actual processes which occur in nature (those which we are trying to model) than flow chart, which is by its very nature sequential. The design phase of these pieces of scientific software was done at a very much higher level than what is traditionally done using only Fortran listings. It was much easier to recognize interdependencies and understand the basic characteristics of the code without wading through detailed listings. The actual coding only comes after the optimizing and debugging are done at this higher level.

During the Implementation Phase the structured English specifications are transformed directly into a high level language such as RATFOR[13], or put into pseudocode, which provides the instructions to a Fortran programmer. The structured English is incorporated directly into the code as comments. In the latter case, the comments are easily stripped out for

documentation by appropriate text processing techniques. In this case, the actual Fortran control statements would be subordinated (right adjusted) to the structured English comments. In the case where RATFOR was used, the actual Fortran control statements would not be present at all.

In using the RATFOR or pseudocode, the four structured programming constructs[11] are used repetitiously. When no precompiler is used, this coding can be done manually from the structured English by technicians and is completely standardized.

When compilation of the various subroutine modules of the code is completed, the available cross-reference listings are used for direct comparison of the variables so listed with the Data Dictionary descriptions of the incoming data flows to the module. Similarly, the Data Flow outputs from the subroutine modules are compared to the cross-reference listings. This was done manually. However, the process could be automated using software tools[13] in the cases here.

The modules are made operational in a Top-Down fashion. Stubs, which perform simple simulations of the actual modules, are used for the lower level modules and are replaced one by one. The particular method of implementation was to incorporate the new subroutine module, call it and immediately dump using a sophisticated memory dump processor. The results of the calculation or functional operation of the routine would be verified. The process was then repeated for other modules during the course of implementation.

During the testing phase, the system as a whole was systematically checked out. This required at least the operation of every single line of coding as well as execution through both sides of every branch existing in the code. The methodology followed that is described in the book on software testing by G. J. Myers.[8] A lot of development work remains to be done in the testing area.

IV. RESULTS AND FINDINGS

Listed in this section are some observations made during the course of the development of three pieces of software by means of Software Engineering. First, it was found that specifications made by means of the Data-Flow-Diagrams, Data Dictionary and Mini-Specs. were a very good means of communicating with the user. They assured him that the ultimate code was indeed going to meet his requirements. These specifications are also extremely helpful to document the code for later understanding by any user.

Following the procedures of Software Engineering leads to almost completely correct code. That is, the final coding can then be trusted to be in a one-to-one correspondence with the original mathematical model of the system. Software Engineering techniques make it very easy to make basic changes in the models after the whole system is up and running. Whole blocks of code are easy to change, as are the specifications themselves. This is because partitioning makes the final coded algorithms much more understandable and much less complex. In addition, good, coherent intermediate executives are created as a matter of course using the methodology, and this also leads to the easy changes.

The incorporation of walkthroughs (peer review) eliminated some very significant bugs at early stages in the development process. In addition, some bugs were found by the walkthroughs in the original MELT code that were not known previously to exist. The actual coding and initial debugging times of the running code were as a result substantially reduced when compared with previous experience.

Software Engineering techniques also made it possible to more effectively utilize highly competent personnel, that is, in doing high level work. Primarily, the latter's efforts are in the Analysis Phase. Lower level technical personnel can do the Design and finally technicians are quite capable of doing the Implementation Phase and the initial testing and debugging phases of the operating code. This overall results in a more effective use of personnel than has ever been accomplished before. It

also greatly reduces need to use senior personnel for routine software maintenance. It should be noted that the tasks identified in the Software Engineering phases can proceed in parallel using differing personnel; this lends itself to a much better organization of the software development.

Finally, a few statistics are presented on the software development using these methods. Presented in Table 1 is the list of some pertinent statistics that were found in the course of this development.

TABLE 1

DISTRIBUTION OF LABOR FOR RESTRUCTURING AN EXISTING CODE

	% of Total Effort
Logic Reformulation	14
Logic Reduction to Structured Programming	17
Data Flow Diagrams	17
Structured English	13
Heirarchy Charts	4
Data Structure Redesign and Implementation	16
Pseudo-code	6
Code	12
Debug	1
TOTAL	100

V. CONCLUSIONS

Our concluding statement is as follows: Application of Software Engineering techniques results in SUPERIOR SOFTWARE. The resulting software is well documented. The documentation is easy to understand (which is very important in the maintenance phase); the software is easy to modify, easier to use and maintain; and is well tested. Finally, the development project will end up with a lower overall cost.

REFERENCES:

1. B. W. Boehm, "Software Engineering," IEEE Trans. on Computers, Vol. C-25, No. 12, (December 1976), pp 1226-1241.

2. G. J. Myers, Composite/Structured Design, Van Nostrand-Reinhardt Company (1978).

3. W. L. Trainor, "Software: From Satan to Savior," Proceedings of the NAECON, (May 1973).

4. E. Yourdon, Managing the Structured Techniques, Yourdon Press, New York City, NY. (1979)

5. E. Yourdon, Structured Walkthroughs, 2nd ed., Yourdon Press, New York City, NY. (1977)

6. T. DeMarco, Structured Analysis and System Specification, Yourdon Press, New York City, NY (Dec. 1978)

7. E. Yourdon and Larry L. Constantine, Structured Design, 2nd ed. New York, YOURDON Press (1978)

8. G. J. Myers, The Art of Software Testing, John Wiley & Sons, New York (1979)

9. E. Yourdon, Techniques of Program Structure and Design. Englewood Cliffs, N. J., Prentice-Hall, Inc., 1975.

10. "Top-Down Programming in Large Systems," Debugging Techniques in Large Systems, ed. R. Rustin (Englewood Cliffs, N. J., Prentice-Hall, 1971), pp. 41-55.

11. E. W. Dijkstra, ED, "Notes on Structured Programming," Structured Programming, O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare (New York: Academic Press, 1972).

12. C. H. Lewis and N. P. Wilburn, MELT-IIIA: An Improved Neutronics, Thermal-Hydraulics Modeling Code for Fast Reactor Safety Analysis, HEDL-TME 76-73, Hanford Engineering Development Laboratory, December 1976.

13. B. W. Kernigham and P. J. Plauger Software Tools, Addison-Wesley Publishing Company, Reading, Mass. (1976).

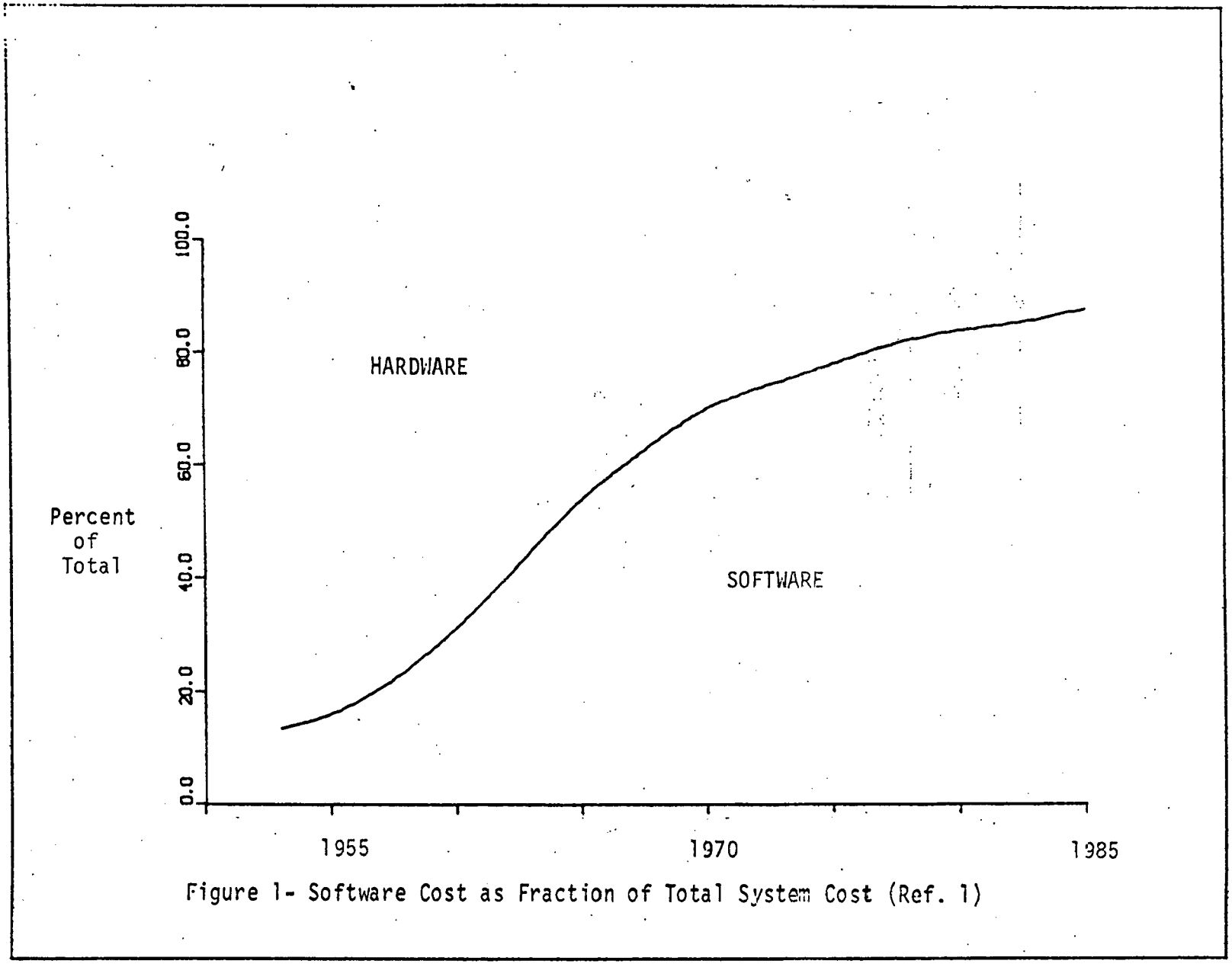

Figure 1- Software Cost as Fraction of Total System Cost (Ref. 1)

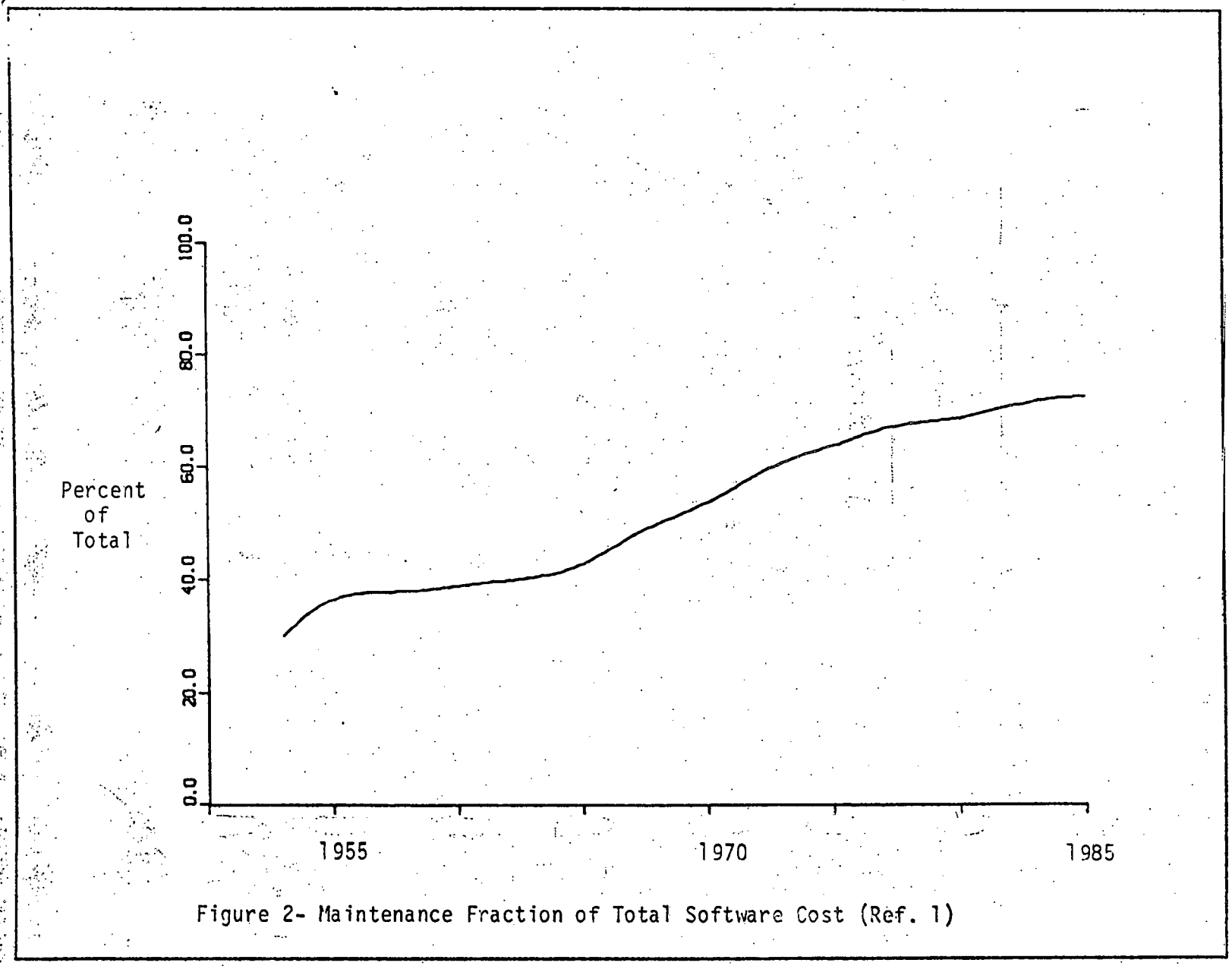

Figure 2- Maintenance Fraction of Total Software Cost (Ref. 1)

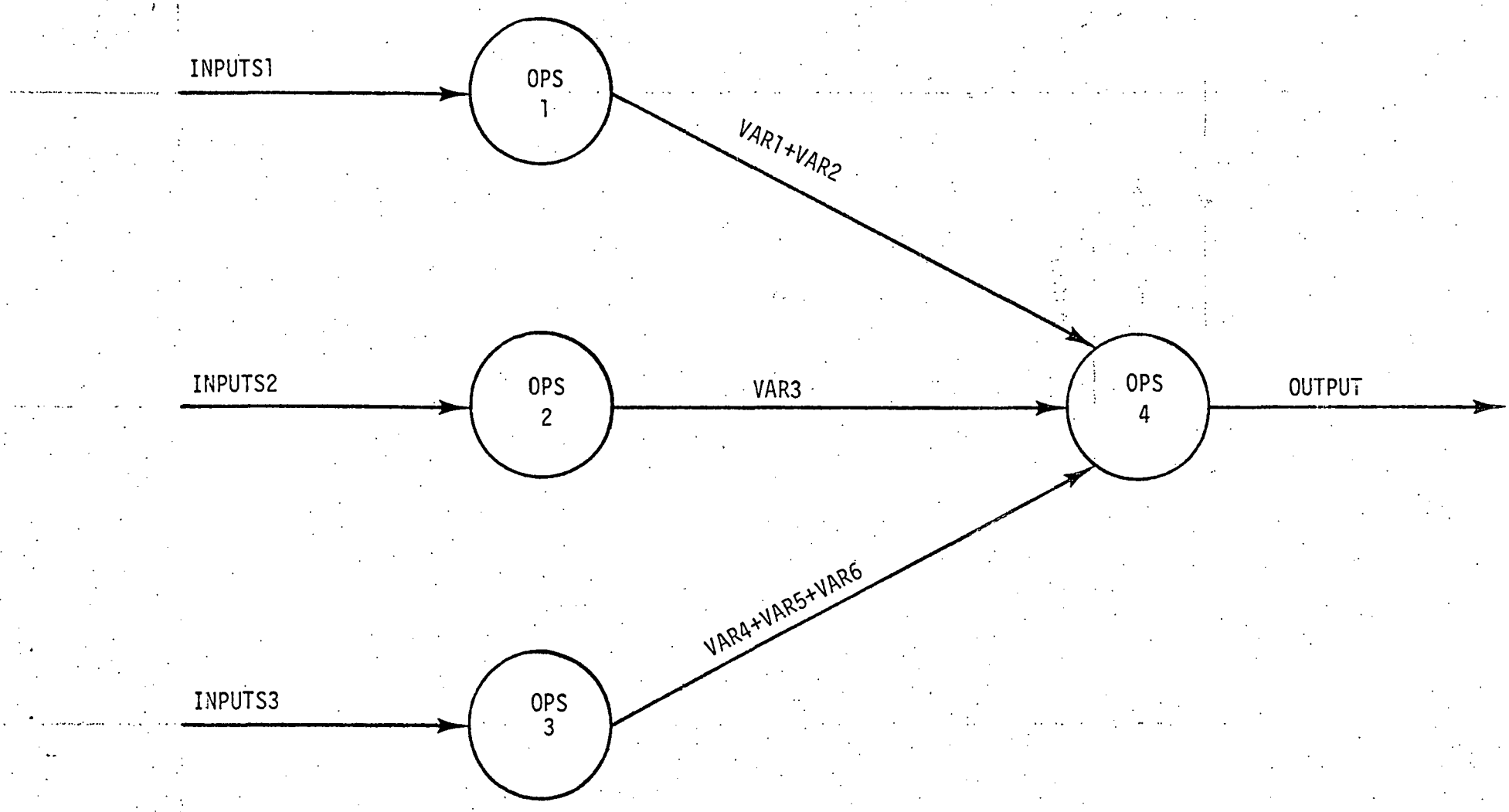

Figure 3. Data flow diagram illustration.

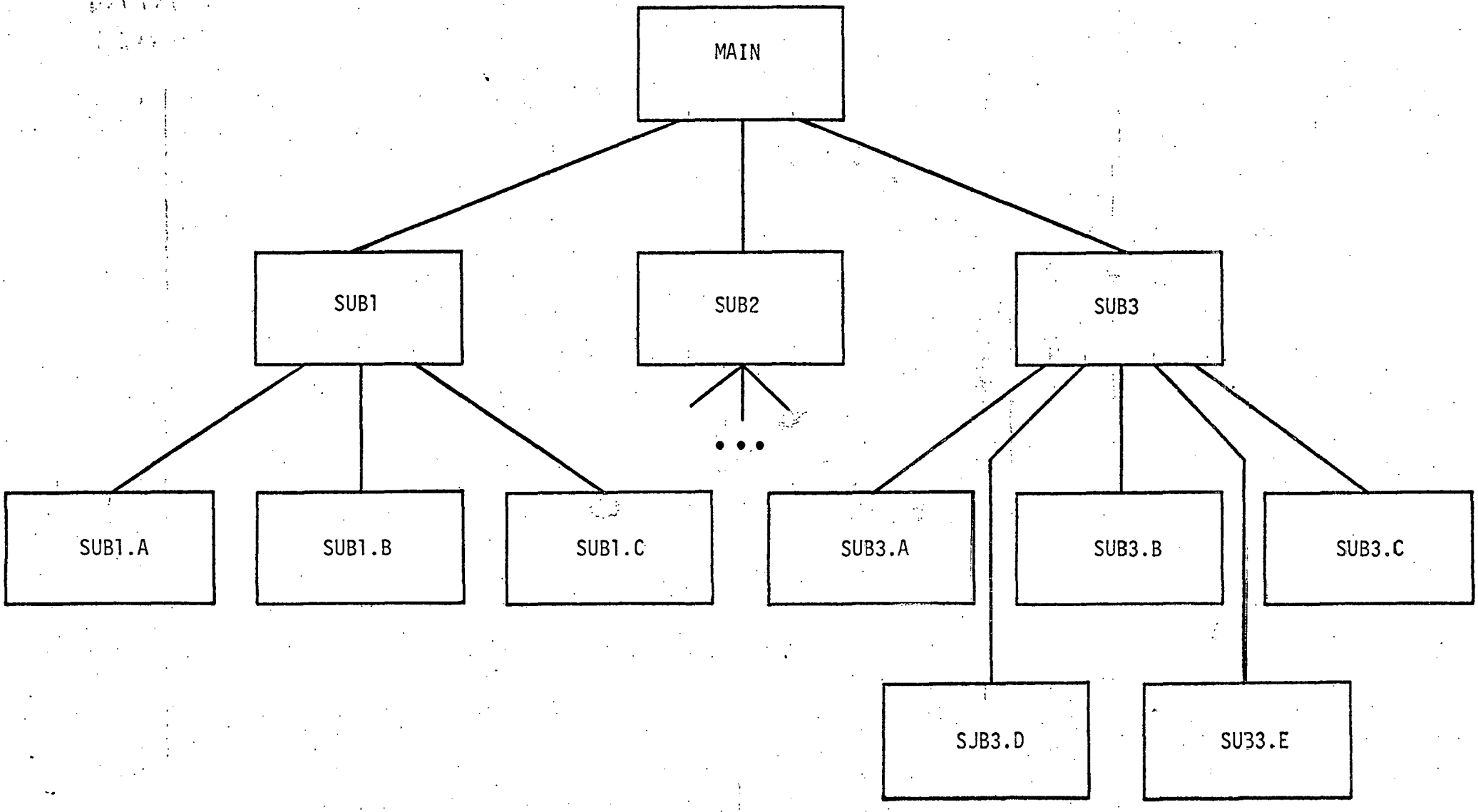

Figure 4. Heirarchy chart

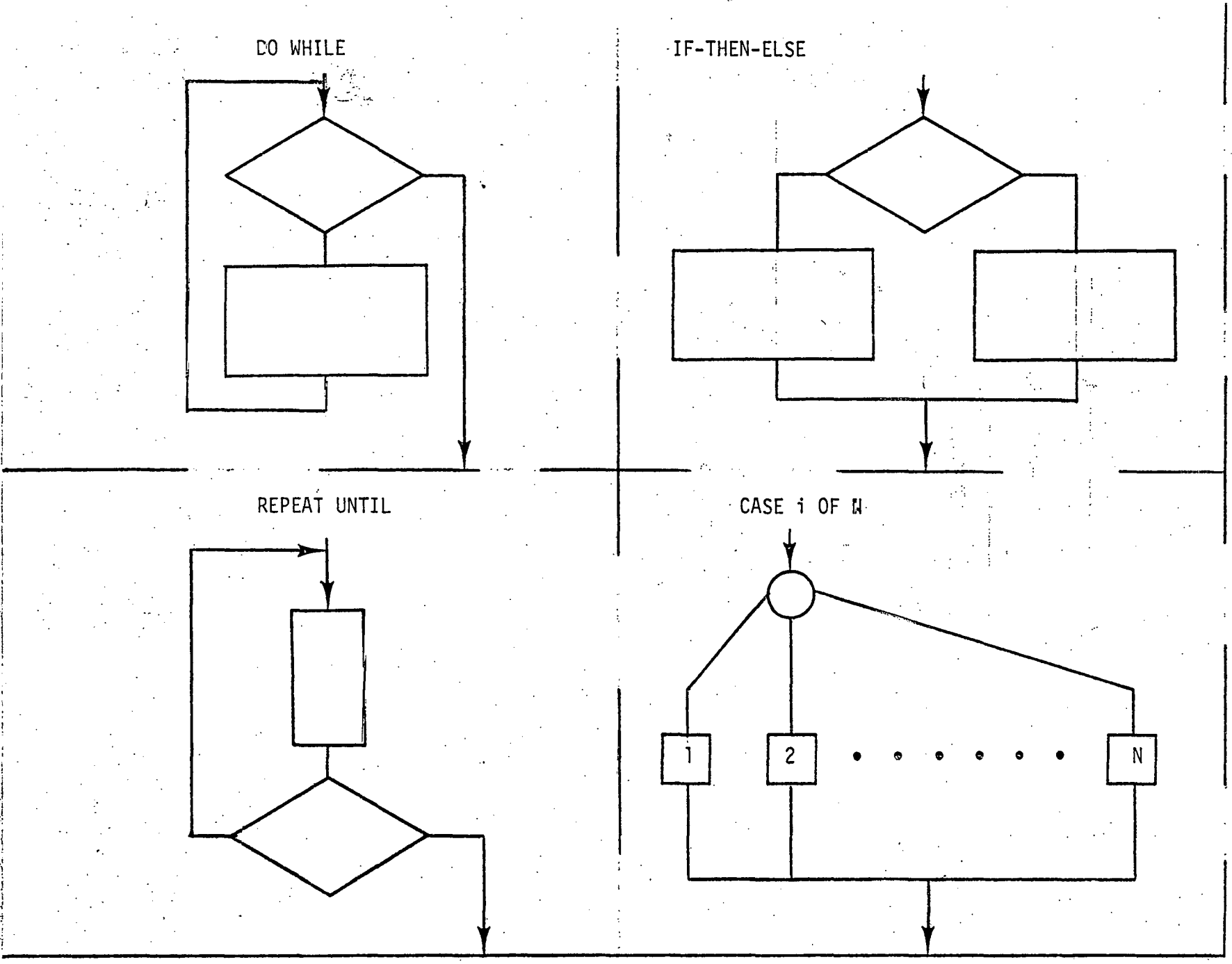

Figure 5. Constructs of structured programming.