STRUCTURED SOFTWARE DESIGN

W.L. WHITE
Honeywell
Marine Systems Division California Center
West Covina, California

INTRODUCTION

A relatively new software development concept is emerging in industry that will lower production costs and increase software product usability. The objective of this paper is to present this new concept to the Navy, so that achievement of an improved, cost-effective software product, available from a variety of procurement sources, may be realized.

In the late 1960's a new term, "structured programming", emerged in the programming industry. Formal papers discussing this subject have been presented by theoreticians well-known to the programming field. Notable among these are Parnas, for his design concepts, and Dijkstra, for his programming language.

The term "structured programming", from my viewpoint, is somewhat of a misnomer; "structured software design" seems rather more appropriate. My opinion in this regard is based on the fact that the word "programming" is limiting in nature, being more often constrained to the equivalence of flow charting and coding. The concept presented in this paper transcends those boundaries. Also included is the presentation of a management solution of the programming problem, in the form of a "chief programmer team" concept.

Since programming is a relatively new field of endeavour, it is naturally to be expected that more rules concerning the development of software products will evolve. Structured design, consequently, is an evolutionary concept representing a positive step in advancing software development toward eventually achieving the status of a science.

Traditionally, the software development process has involved a cascading series of interfaces, beginning with the creation of individual modules, selecting two for interfacing (after debugging), and checking out the interface; with the first two checked out and working, a third is interfaced with them, and so on until the program is built. This is usually described as the "bottom-up" approach, in which a cycling program is not realized until the final module has been interfaced. Thus the total program has been cycling for only a short period prior to hardware/software integration.

Although much is to be said for the use of structured software design, it should be remembered that it is not a panacea. It is, however, a positive step in the direction of reduced programming costs and increased understandability of the product.

STRUCTURED SOFTWARE DESIGN

As it has emerged in industry today, structured software design concerns itself with several aspects of programming, constituting an amalgam of several elements: (1) specification development (2) a top-down design approach (3) modularity (4) a design language and (5) the chief programmer team concept.

SPECIFICATION DEVELOPMENT

A computer program performance specification is an essential first step required for the top-down design approach. This document specifies all of the things that the program is going to be required to do, without specifying how they will be done. Next, a computer program design specification is prepared, specifying how the program will implement each of the computer program performance specification requirements. The design of the program strives to maintain a hierarchal structure, using a top-down implementation. It is very important that the performance specification contain a contract correlation matrix, permitting the tracing of contractual requirements into that specification. It is necessary, furthermore, that each requirement be traceable into the design specification. The design specification matrix shows the allocation of given requirements to a specific module or modules.

TOP-DOWN DESIGN APPROACH

In the top-down design approach, the first module to be prepared is the Executive. Then a system of "dummy modules", consisting of stubs (or tabs) representing the existence and location of the other program modules, is prepared.

Once the Executive is debugged, it, together with the dummy modules, constitutes a cycling skeletal program.

As individual modules are prepared and debugged, they replace their corresponding stubs in the structure. This process provides a continuously cycling program, yielding continuous program development progress as more and more modules enter the program. At the time that the last module is inserted, the earlier modules will have been cycling for a significant period in advance of hardware/software integration, a factor that works to enhance credibility of the program product.

MODULARITY

Since a module with a single function will be more easily understood by programmers, the unifunctional module is preferred to the multifunctional. The unifunctional module is more easily debugged, and, in general, will provide a computer program that is intellectually manageable. Using single function modules will help to provide low module interdependence, with a minimum of interconnections required. This approach will result in programs economical to produce, more reliable, i.e., fewer latent defects, and definitely more manageable.

DESIGN LANGUAGE

Structured software design proposes that flow charts are no longer required. Observation of program design activities over a number of years indicates that, in spite of instructions to the contrary, programmers will commence coding of a program prior to flow chart completion. Investigation of this phenomenon reveals that the coding language provides the programmer with a significantly more powerful design tool than that provided by the flow diagram. Consequently, a design language is proposed in lieu of flow charting. The design language is a high order language, and may be provided with or without an associated compiler. Programmers, working as individuals or in groups, can define the language to be used on a given project. Once the program has been designed in the selected language, translation to the assembly or compiler language must be made. In the case where a compiler already exists to bridge this gap, the programmer's task is complete; otherwise he must make a manual translation.

An example of the use of a program language is presented in Attachment 1.

Additional Approaches to Design

Some additional approaches, "chunking", "abstraction", and "information hiding", are involved in the design area to support the program's intellectual manageability.

Chunking

It has been determined scientifically that the human mind gathers information and retains it in chunks. For example, consider your telephone number, consisting normally of 10 digits. To remember the number your mind breaks it into three identifiable chunks: a three-digit area code, three leading digits, and four terminal digits. This concept can be applied to the development of a computer program so that the tasks that the program is to perform are chunked in a logical manner and thus provide a more *manageable design.*

Abstraction

The concept of abstraction as defined by C.A.R. Hoare, is somewhat different, in that it involves the decision to concentrate on properties shared by many objects or situations in the real world, and to ignore the differences between them. For example, when you visit a housing tract you usually think of the elements as houses, without distinguishing in your mind the differences between each and every house. In this way, you are able to consider in the word "house" any element of that set, each of which in reality is different. For a programming example, consider an array of 24 bits packed into one word, and an array of 8-bit items packed two entries per computer word. The abstraction array applies to each of these, but the detailed representation is clearly different. In designing a computer program, the use of a hierarchal structure, containing different levels of abstraction, is recommended. The recommended structure will provide that information processing occurs from top to bottom. The design will assure that a module at one level will be totally ignorant of the existence of modules at higher levels. Any given module will know only about modules at lower levels. For example, a high level module may be designated to process motion. The abstraction of motion will be broken down (chunked) into that required for targets, weapons, etc. Each chunk then will be relegated to lower level modules.

Information Hiding

D.L. Parnas has an interesting concept which he calls "information hiding". This concept says that a module would know, exclusively, a given segment of design information. For example, in most training device applications, the real-time program has an output table which is delivered to the interface at some regular frequency. Various computational modules place information into the output table. Changing the format of the output table may require changes in some or all of the computational modules. Use of information hiding will permit a change in the format of the output table with no change required to any computational module. This will be accomplished by having each computational module communicate to an output module where only the output module knows the format of the output table. This is a valuable concept in that when a change is made to one area of a program, it frequently tends to produce a ripple effect; information hiding can reduce or virtually eliminate the ripple effect.

CHIEF PROGRAMMER TEAM

As mentioned earlier, the chief programmer team concept is an important cornerstone of this approach. This concept requires that for a given programming project there is a chief programmer who is responsible for the basic architecture of the system together with a fully qualified backup programmer. In addition, there is a programming secretary. The secretary's duties include keeping all project records, all test data and results, and up-to-date documentation for the project. The secretary is able to free the programmers so they may concentrate on the design and implementation. The team concept also employs "egoless" programming. Egoless programming is discussed by Weinberg[1]. Use of egoless programming provides a more unified team approach, reduced time in checkout, and better software designs. Egoless program has evolved from the consideration that normally a programmer does not care to have another programmer examine his design or coding until he has it checked out. He does not care to share flaws or errors he produced. Egoless programming requires that each programmer pass his design around to other members of the team. This will permit evaluation by others and the discovery of design flaws in advance of implementation. It is normally cheaper to correct a flaw during design than during checkout. Each programmer should have a backup programmer who is as familiar with the module being developed as the originator. The programming group secretary will maintain test data results as a matter of public record. As one programmer's test data begins to grow by the foot where another's appears to grow by the inch, it quickly becomes obvious who is the weak member of the team. From this it becomes apparent that team members will weed out the weaker members by comparison. Time will not be wasted in an effort to correct and support flaws of weaker individuals.

In the classic definition of structured programming, the implementer is constrained to use only three basic operations. These are the simple process operations, the do-while-if loop, and the decision loop (see Figure 1). It is believed that any computer program may be produced using these three, and only these three operations. It should be noted that this eliminates the use of the unconditional go-to-statement. The merits of this limitation will not be discussed here, since it is a subject unto itself. It should be pointed out, however, that I believe a program without the use of unconditional branches would be a "better" program; however, I believe that there are circumstances under which one should allow the use of such instructions. In addition, I believe that one should allow a fourth operation, subordination. There will be many cases where the memory constraint will require the use of unconditional branches and subroutines.

In conclusion, structured software design is intended to reduce software production costs, increase the intellectual manageability of the end product, and provide higher reliability. The approach requires specification of requirements, design from top down, team organization, and use of a design language.

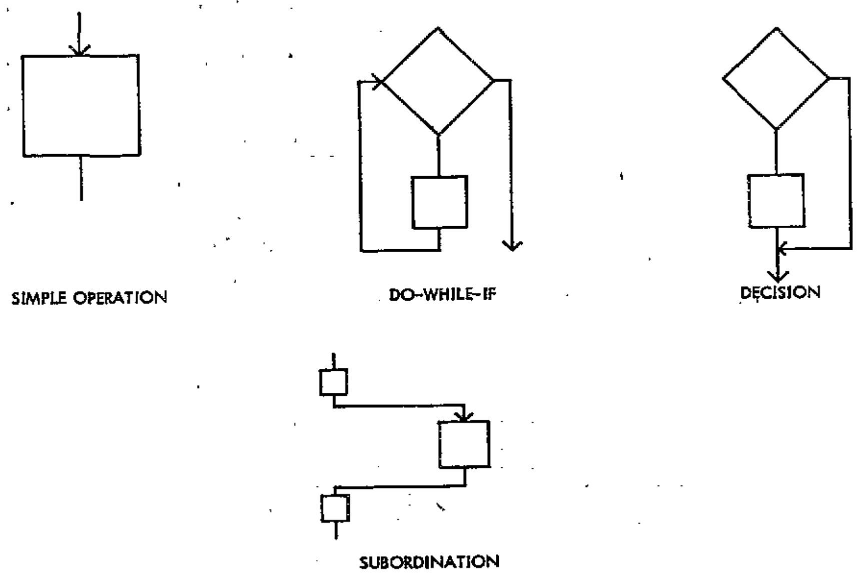

Figure 1. Three Basic Process Operations

[1] Weinberg, Gerald M. The Psychology of Computer Programming, New York, VanNostrand Reinhold (1971)

ATTACHMENT 1

DESIGN LANGUAGE EXAMPLE

For this example, consider the problem to require solving for the hypotenuse of a right triangle, (a, b, c), or extracting the cube root of the sum of the squares of the same triangle. The cube root function will be executed if the value of Z = 1.

Use of a design language would yield the following:

If Z = 1, then

$(a^2 + b^2 + c^2)$ 3

End

Otherwise,

$c^2 = a^2 + b^2$

End

The above must now be put into a source language which can be translated into a machine language. In the example, a manual translation is required. If ALGOL has been used, an automatic compilation could have been used.

The flow chart design approach, Figure 1-1, could yield:

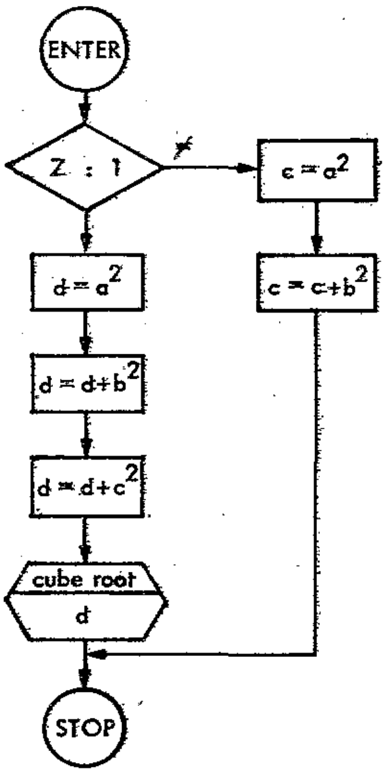

Figure 1-1. Flow Chart Design Approach

ABOUT THE AUTHOR

MR. W. L. WHITE received his B.A. degree in Mathematics from the University of California at Los Angeles in 1962. For the past 13 years, Mr. White has been affiliated with Honeywell's Marine Systems Division California Center where he is Program Project Supervisor. His experience includes 2 years as Group Leader of Real-Time Programs, activities in such key projects as the Navy's DD-963 Program which involves program development for Underwater Fire Control System MK 116 and Air Force Air Systems Division/Air Training Command Undergraduate Navigator Training System (UNTS). In this project, his group developed simulation programs for a 52-station, ground-based simulator to train Air Force Navigators.

PAPERS PUBLISHED, BUT NOT PRESENTED