3

on structured programming

"In the light of all these difficulties the simple approach may be the best."
W.F.C. Purser (1976)

Software for local multi-mini processor systems.
In: 'Multiprocessor Systems', Infotech State of the Art Report, pg. 408.

Contents Chapter Three

CHAPTER THREE

ON STRUCTURED PROGRAMMING

3.1. INTRODUCTION

One conclusion we can draw from the discussion in chapter two is that correctness proving is by no means a trivial task, neither for sequential programs nor for concurrent programs. The available analysis tools are not powerful enough to allow for the verification of just any program. By necessity one must restrict oneself to simple and well-structured programs.
But, one may ask, what is a well-structured program?
Fortunately, there is a wealth of literature on this subject. The larger part of this literature is primarily concerned with sequential programs, but still we should take careful notice of these theories if we want to be able to apply the underlying principles to the multiprocessing cases.

In this chapter we will study the techniques which have become known as 'strucured programming' techniques. Before doing so we will dicuss the origination of the theory and in that context we take a brief look at the so-called 'software crisis' which stimulated this origination in many ways. We will also consider how the structured programming methodology is related to more general problem solving techniques. We will then survey the principal techniques from structured programming, while concentrating ourselves on these four major aspects: restrictions, abstraction, stepwise refinement, and clear notation.
We conclude with a discussion of some of the major criticisms that have been raised against the techniques.
It is not our intention to give, within the scope of a single chapter, a detailed overview of all techniques that have been brought in connection with structured programming theory. Such overviews can be found elsewhere (see, for instance, **Infotech '78**). Our present goal is to describe and motivate the main principles, in preparation of the studies in part 2 and 3 of this thesis.

3.1.1. The origination of structured programming

The term 'structured programming' was coined by **Dijkstra '69b**. Though Dijkstra is not the sole originator of the fundamental ideas behind structured programming, he certainly was the first who succeeded in formulating his ideas on the subject in a way which was understood and which initiated a wide discussion. Without exaggerating, it can be said that Dijkstra created a new paradigm in the computer sciences **(Kuhn '62)**.
In retrospect, the occurrence of a 'paradigm shift' at that moment (i.e. the late sixties) is easily understood. One was in the middle of the 'software crisis'. Computers had become faster and bigger than anyone could have foreseen, and programmers were confronted with unmanageably large and complex software. The construction of a reliable and understandable program of some size had become an art, rather than a teachable skill. It was noted that the complexity of a program grew almost exponentially with its size.

The negative effect of the use of certain language constructs on the understandability of a computer program had been debated on a small scale for some

years. Some initial thoughts on the subject can be found already in **Dijkstra '61** (especially **pg. 4**).

> In **Dijkstra '68a** a reference is made to remarks of Zemanek in 1959. **Knuth '74 (pg. 264** and **265)** refers to discussions with Schorre in 1963, to an article of Naur in '63 and to papers of Landin and Naur presented at an ACM Conference in 1966.

The discussion was however only really started after the publication of Dijkstra's provoking letter '*GO TO Considered Harmful*'. The opening sentences of that letter clearly illustrate its purport and style:

> "For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of GO TO statements in the programs they produce. More recently I discovered why the use of the GO TO statement has such disastrous effects, and I became convinced that the GO TO statement should be abolished from all 'higher level' programming languages (i.e. everything except, perhaps, plain machine code)."
> **(Dijkstra '68a, pg. 147).**

This letter unleashed a vast amount of articles, conference papers, and emotional discussions. Another influential article of Dijkstra's was published a few months later **(Dijkstra '68b)**. In that paper Dijkstra illustrated how a large complex software system could be designed and programmed in 'layers', by systematic abstraction. One year later, in 1969, the '*Notes on Structured Programming*' appeared as a report of the University of Technology in Eindhoven **(Dijkstra '69b)**. Despite the fact that these 'notes' were not distributed on a wide scale, they had a large influence. In 1971 Wirth elucidated one of the design techniques, 'stepwise refinement', to a wider audience **(Wirth '71)**.
One year later, in 1972, Dijkstra's notes were re-published in book-form, supplemented with papers by Hoare and Dahl **(Dahl, Dijkstra & Hoare '72)**.
Two other important articles must be mentioned here. Both appeared in 1973. The first one is the article by Wulf and Shaw titled '*Global Variables Considered Harmful*' **(Wulf & Shaw '73)**. (The titles of Dijkstra's articles have been paraphrazed in many ways.) Wulf and Shaw argued that the use of variables with a large scope can frustrate the analysis of a program. These ideas were later elaborated by Dijkstra **(Dijkstra '76, pg. 79-93)**.
The second article is the one from Nassi and Shneiderman who introduced a new flow-charting technique for structured programs **(Nassi & Shneiderman '73)**. This technique, which was aptly named 'Structogramme' by **Schnupp '74**, derives its value from the fact that it forces a programmer to think in terms of only a few basic control-flow structures and prevents the (undisciplined) use of GO TO statements.

We can summarize the principal ideas from the papers mentioned up to now in these four concepts: restrictions, abstraction, stepwise refinement, and clear notation. Each of these items was heavily debated. The ideas were extended, modified and reduced, they were defended, ignored and attacked in large waves of articles.

Of considerable influence upon the eventual acceptance of the ideas were the many publications on the successful application of the techniques in large industrial programming projects. Already in 1973 there appeared an article entitled '*Revolution in Programming*', from which we quote:

> "Then came the IBM work for the New York Times, with reports of greatly increased programmer productivity and very greatly reduced coding error rates (one detected error per 10.000 lines of coding, or one per man year)! Absolutely incredible, but these were the facts."
> **(McCracken '73, pg. 51).**

Evidently, the reportings were often called in question. For instance, one noted that successes are published sooner than failures. Recently, a large software institute wrote skeptically in a customer letter:

> "- Many of the early successful projects were run by experienced organizations using top-level staff - how do the new techniques work in practice in everyday installations?
> - The very fact that an organization attempts to introduce new techniques suggests a commitment to improving software production - how much of the reported improvement is due to that commitment, and how much to the new techniques?"
> **(Infotech, letter R377, Nov. 1978).**

A more practical problem was that in many cases the programmers involved in the industrial projects simply refused to alter their style of programming (see **Canning '74, pg. 3; Gries '74**). The structured programming principles were criticized both on practical grounds and for more philosophical reasons. We take a closer look into the concrete criticisms in a later section (see section 3.4). First we take a closer look into the nature of the 'software crisis' and on general problem solving techniques.

3.1.2. The software crisis

In 1966 Naur wrote:

> "The stress has been on always larger, and, allegedly, more powerful systems, in spite of the fact that the available programmer competence often is unable to cope with their complexities.
> However, a reaction is bound to come. We cannot indefinitely continue to build on sand. When this is realized there will be an increased interest in the less glamorous, but more solid basic principles."
> **(Naur '66, pg. 310).**

Within a few years Naur's prediction came true. In 1973 Donaldson already described 'a major shift of emphasis' in software engineering.

> "The primary requirement to be met in software engineering has always been to perform the function specified for the software. But, where at one time secondary emphasis was placed only on software efficiency, that is in core and time required, today three other factors are recognized as requiring special emphasis. These factors are reliability, maintainability, and extensibility."
> **(Donaldson '73, pg. 52).**

We can point out four major reasons for the occurrence of the change described by Donaldson:
First of all, the machine is today no longer the most expensive 'link' in the programming chain. Computer time has become relatively cheap. (See chapter 1, section 1.1.)
Secondly, the considerable growth of computers and programs, both in size and complexity, has made their reliability more and more important. Software simplicity has become more important than hardware efficiency.
Thirdly, the average life time of software has increased. Existing programs must be adaptable to changing requirements, and not only by the people who actually wrote the programs in question. There are many ways to write a program which no one understands **(Kernighan & Plauger '74)**, but to write programs which are transparent to any programmer is much more difficult. It requires another attitude from programmers and a style of programming which Weinberg once described as 'ego-less programming' **(Weinberg '71)**.
The *fourth* reason concerns the multiprocessing systems. The advantage of a mul-

tiprocessing system is no longer the fact that it is more 'economical' to share hardware equipment among many users, but it is the possibility of sharing information. To guarantee the integrity of this shared information has become a major problem **(Update, Vol. 2, No. 6, June 1977)**.
These observations lead unavoidably to the conclusion that concerns for machine efficiency should no longer be a guiding principle in software design. Garren formulated it even more sharply, when he wrote:

> "As a general rule local efficiency, i.e. bit twiddling, is in conflict with the need for understandability (-)."
> **(Garren '73, pg. 37)**.

This does however not imply that the concerns for efficiency are to be entirely neglected. Garren continued:

> "By emphasizing ease of modification in system design and implementation it is possible to quickly build and debug the system and then measure it to discover where optimization is actually needed."
>
> "Local optimizations can contribute to a relatively small fractional increase in program efficiency, whereas changes in global algorithm can result in savings of orders of magnitude."

Concerns for program efficiency are thus merely moved to the final stages in program design (see especially **Knuth '74**).

3.1.3. Structured programming and problem solving

After the software crisis it became clear that the real bottle-neck in software-development was not 'efficient coding' but structuredness, or 'efficient problem solving'. However, as Gries aptly remarked:

> "... almost none of the elementary programming books say anything about problem solving."
> **(Gries '74)**.

It is one of the merits of structured programming that it filled this gap by constructing an analogon of known problem solving techniques in program design theory. Dijkstra explicitly described the task of writing a correct computer program as an intellectual problem:

> "The art of programming is the art of organizing complexity, of mastering multitude and avoiding its bastard chaos as effectively as possible."
> **(Dijkstra**, in: **Dahl, Dijkstra & Hoare '72, pg. 6)**.

To design a program is to determine its structure. But, not just any structure will make a program understandable. Especially, the GO TO statement allows a programmer to make highly complicated control-flow patterns.
Dijkstra suggested therefore that the programmer restrict himself mainly to program structures which correspond closely to standard reasoning patterns. When one deletes the GO TO statement, the programmer has only the following 'tools' for the construction of such patterns: concatenation, conditional selection, iteration, procedure-calls, and recursion. Procedure-calls can (conceptually) be replaced by the code they represent and they are therefore not a fundamentally different type of tool. A recursion can be represented or understood as an iteration, and it is therefore usually not considered separately either. That leaves us with: concatenation, selection, and iteration. Liskov writes:

> "... each permissible control structure is associated with a well-

known rule of inference: concatenation with linear reasoning, iteration with induction, and conditional selection with case analysis."
(Liskov '72, pg. 194).

A well-structured program then should be decomposable into these three basic structures. Ideally this decomposition should be equivalent to a decomposition in 'logical units' of the problem being solved by the program.
In this manner even large programs should be understandable with an effort which is at most *directly* proportional to their sizes, instead of exponentially proportional. The major problem in programming is, however, not to write a program with only the three basic control-flow structures mentioned, but the problem is to make the structure of the program 'match' the structure of the problem being solved. To solve *that* problem the programmer is advised to use elementary problem solving techniques like abstraction and stepwise refinement. Knuth wrote:

"We understand complex things by systematically breaking them into successively simpler parts and understanding how these parts fit together locally."
(Knuth '74, pg. 291).

By systematic abstraction one can attempt to distinguish minor aspects (implementation details) from major lines, and concentrate on the latter. By systematically splitting up major problems into minor ones, one can derive a solution in a stepwise manner.

The ideas are not new **(Knuth '74, pg. 291/292; Lecarme '74).** It has even been reported that some of the techniques had been in use for some time in 'artificial intelligence' projects **(VandenBrug '74).**

Top-down or bottom-up.
One of the more heavily debated questions with respect to the structured programming methodology is whether programs should be designed 'top-down' or 'bottom-up' (e.g. **McClure '75).** One of the early writers on problem solving, Polya, says:

"We have before us an unsolved problem, an open question. We have to *find a connection between the data and the unknown.*
We may represent our unsolved problem as an open space between the data and the unknown, as a gap across which we have to construct a bridge. We can start from either side, from the unknown or from the data ..."
(Polya '48, pg. 73).

Polya did not express any principal preferences for either of these approaches, and indeed it would be unwise to do so. In problem solving and in program design both analysis and synthesis play an important role. It has often be noted that problem solving is essentially an iterative process. (See, for instance, **Davies '71, '76).** It consists of repeated attempts to match the results of analysis and synthesis. Similarly, the authors of the book '*Structured Programming*' **(Dahl, Dijkstra & Hoare '72)** explicitly stated in their preface that the:

"... structured programming principles can be equally applied in 'bottom-up' as in 'top-down' design."

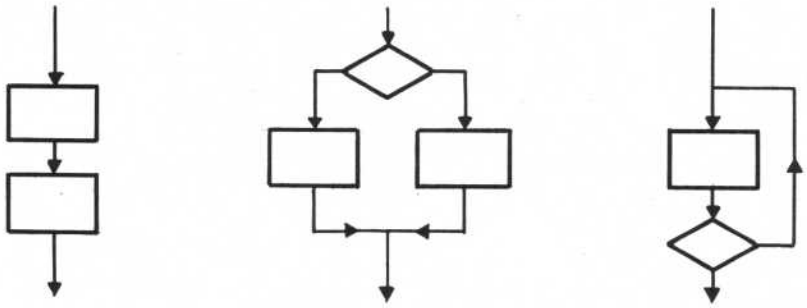
Still, structured programming is often identified with purely 'top-down' design. The confusion is perhaps caused by the fact that one of the requirements of structured programming is 'top-down documentation'. Documentation, however, concerns completed programs and not primarily programs which are still being designed. For a further discussion of these points we refer to the section on

'criticisms' (section 3.4). We will now give a survey of the principal structured programming techniques. We will do this under the headings: Restrictions, Abstraction, Stepwise refinement, and Notation.

3.2. SURVEY OF STRUCTURED PROGRAMMING TECHNIQUES

3.2.1. Restrictions

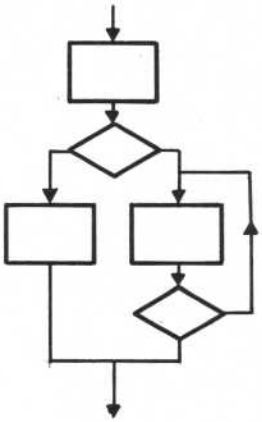
The most important restriction is that the programmer is asked to use only three basic control-flow structures: *concatenation*, *conditional selection* and *iteration*. These three basic structures are to be used as the 'building blocks' of a program. They all have one entry-point and one exit-point, as illustrated in figure 3.1[1].



(a) *Concatenation* *(b)* *Conditional Selection* *(c)* *Iteration*

Figure 3.1.

To form larger structures, the three basic blocks may be nested in any order, as illustrated in figure 3.2.



Nestings

Figure 3.2.

The program which is designed in accordance with these restrictions consists of a hierarchical nesting of control-flow structures, and has precisely one entry-point and one exit-point on each level. It can then always be established, with little effort, *when* a certain program part is executed and *why*.

It is of course not true that every goto-less program is by necessity transparent, nor that every program *with* goto-statements is unclear. Wulf argued:

> "... it is certainly possible to find programs which use the goto and are not difficult to understand - particularly if the programs are small, the targets of the goto's are close to the jump points, and the control paths established by the goto's use are standard ones."
> **(Wulf '73, pg. 28).**

Two 'standard cases' in which the use of a goto statement may be hard to avoid, are the 'search-loop' and the 'error-exit' (see also section 3.4).

- - - - -

(1) We will derive a complexity measure for control-flow structures in appendix B.

Wirth, however, commented:

> "... often the need for an *exit in the middle* construct is based on a preconceived notion rather than a real necessity, (-) sometimes an even better solution is found when sticking to the fundamental constructs."
> **(Wirth '74, pg. 255).**

There have been many proposals for the inclusion of a special construct in the 'set of basic control-flow constructs' which would cover the error-exit and/or the search-loop (e.g. **Knuth & Floyd '71; Friedman '74; Evans '74; Goodenough '75**). There is, however, no consensus on this point. More liberal authors have suggested not *eliminating* the goto statements, but *restricting* their usage. Three alternatives are then:

(1) to restrict to only *forward* goto-jumps;
(2) to restrict to goto's used for error-exits only;
(3) to restrict to goto-jumps *within the innermost hierarchical level*.

Each variant (and combination) has found its advocates (see, for instance, **Leavenworth '72; Canning '74; Newton '75**).
While some authors have suggested *relaxing* the restrictions of structured programming, others have proposed still more stringent rules.
The most important of these are Wulf and Shaw **(Wulf & Shaw '73)**, and Hoare (in: **Dahl, Dijkstra & Hoare '72**).
Hoare stressed that not only the control-flow of a program should be well-structured but also its data. In the process of stepwise refinement the programmer should choose a plausible data representation and formalize the necessary data manipulation techniques. For instance, the programmer can choose to represent his data in a structured record, a queue or a list, and define the appropriate amending, rearranging, and deleting operations for it. As special data-types Hoare discussed the 'cartesian product', the 'discriminate set union', the 'power set', and the 'sequence'. We consider some other ideas on data structuring more closely in the section on **Notation** (3.5.4).

Wulf and Shaw suggested restricting the scope of variables, and requiring that this scope is always made explicit in a program text via declarations. The reasons for these further restrictions are:

> "... the non-local variable is a major contributing factor in programs which are difficult to understand."
>
> "... the complexity of [**the task to prove that variables are always accessed in the intended way**] increases exponentially with the distance of the last previous use of each variable along each control path."
>
> "... the complexity of the abstraction is directly related to the number of non-local variables used."
> **(Wulf & Shaw '73, pg. 28/29).**

Wulf and Shaw further argued that an access right to a variable which has been declared on a high level in a nesting hierarchy should not be 'inherited' (exported) automatically by (to) all lower levels . Upon entry to a new level (a 'block') all variable declarations that are to be 'inherited' (imported) must then declared be explicitly. Dijkstra elaborated similar ideas in **Dijkstra '76**.
An inherited (imported) variable is called 'global' by Dijkstra, a new variable is called 'local'. The scope of a local variable extends from the begin-

ning to the end of the block in which it is declared, with the exception of those inner blocks that do not explicitly inherit it via a 'global' declaration **(Dijkstra '76, pg. 85)**. All variables must be initialized before their first use. A non-initialized variable is called a 'virgin' variable. Via the declarations one can declare whether the variables will be used as references (constants) or will be changed in the corresponding block. Dijkstra then defines 6 different types of variables which can occur in a block (local, global or virgin variables, used as constants or as 'true' variables) with special 'rules of inheritance'. **(Dijkstra '76, pg. 91)**.
As a possible extension of these ideas both Dijkstra and Wulf & Shaw have mentioned the possibility of making explicit and restricting also the ways in which variables can be modified.

Another type of additional restriction was suggested by **Bloom '75** and **Weinberg** *et al.* **'75**. They suggested eliminating the 'conditional selection' construct, and replacing it by a 'conditional execution (i.e. replacing the 'if-then-else-end' by 'if-then-end'). This suggestion has not found much support, and indeed it seems that it would be too restrictive. It would make program structures more simple, but it can be questioned whether it would make them more understandable.
An article which is often quoted in discussions on structured programming techniques is **Bohm & Jacopini '66**.
Bohm & Jacopini '66 showed that every flow diagram with goto-statements can be transformed in a goto-less one. (See also **Mills '72** who presented a more rigorous proof of their statements. See also **Ashcroft & Manna '71.**) The programs are then equipped with some additional state variables which formalize part of the flow-of-control. It will be obvious, however, that such transformations have no relevance to structured programming whatsoever. Dijkstra stated:

> "The exercise to translate an arbitrary flow diagram more or less mechanically into a jumpless one, however, is not to be recommended. Then the resulting flow diagram cannot be expected to be more transparent than the original."
> **(Dijkstra '68a, pg. 148)**.

The same point was also clearly illustrated by Cooper who demonstrated, in a reaction to Bohm & Jacopini's paper, that any flow diagram can trivially be transformed into a single iteration loop containing one large case-selection structure. The resulting program is 'simple' indeed, but certainly not understandable **(Cooper '66)**.

3.2.2. Abstraction

In the initial stages of program designing the main problem is to find the major lines of the solution. Abstraction (or 'generalization' as Polya called it) is a powerful mental aid to discover what is detail and what is not.

> "Executional abstraction is so basic to the whole notion of 'an algorithm' that it is usually taken for granted and left unmentioned. (-) ... it refers to the way in which we can get a specific computation within our intellectual grip by considering it as a member of a large class of different computations ..."
> **(Dijkstra '76, pg. 1)**.

The abstraction levels can be expressed in a program structure in several ways. First of all, one can express it in the *ordering* of the control-flow structures in a nesting hierarchy. The flow-structures on the highest (outer) levels in the nesting hierarchy should then express the major lines in the argument. Secondly, one can express it in the subdividing of the basic computational

functions used in the program, and in their relationships.
The technique of dividing the logic required of a program into pieces (modules) which are small, simple and independent is often called 'modular programming' **(Harding '72)**. One can then construct a 'module hierarchy' in the structural relationships of these modules. Major lines are treated in 'high level' modules, details are treated in the lower level ones. It is however not intended that the entire program should be constructed as an arbitrary collection of modules and module-interfaces, at least not in structured programming. In Dijkstra's view a large program should be constructed in 'layers' **(Dijkstra '68b, '69a, '72)**. Each layer represents an abstraction level. On a high level of abstraction one works with abstract operations and functions which are elaborated in the lower levels (see chapter 1).
The concepts of a 'module' and a 'layer' are closely related though. Dijkstra explained the differences as follows:

> "If we succeed in building up our program along the lines just given, we have arranged our program in layers. Each program is to be understood by itself, under the assumption of a suitable machine to execute it, while the function of each layer is to simulate the machine that is assumed to be available on the level immediately above it."
>
> "The picture of a layered hierarchy of machines provides a counter poison to one of the dangers evoked by ruthless application of the principle 'Divide and Rule', viz. that the different components are programmed so independently of each other that duplication of work (or worse) takes place.
> The fact that a layer contains 'a bunch of programs' to be executed by some conceptual machine stresses the fact that the programs of this bunch are invited to share the same primitives. Separation of tasks is a good thing, on the other hand we have to tie the loose ends together again."
> **(Dijkstra, in: Dahl, Dijkstra & Hoare '72, pg. 49/50).**

Another concept which is related to the principles of 'abstraction' and 'layering' is 'information hiding'. Here one makes use of the hierarchical structure of a well designed program. The basic idea is then to hide information about the implementation details of the lower levels from the higher levels. This information hiding serves two purposes: integrity and adaptability. By hiding information about the way in which a memory-element is listed in an abstract queue, for instance, one can prevent the information contained in that queue from being mutilated by an erroneous usage. Similarly, if one hides information on the way in which specific design decisions are locally solved to the larger part of a program, it will be a trivial task to adapt that program to alternative design decisions **(Parnas '72; Dennis '75; Ross *et al.* '75)**.

3.2.3. Stepwise refinement

The process of stepwise refinement can be described in general terms as follows:
A programming problem is first analyzed thoroughly to get an overview of all the implications of the problem, without however going into specifics.
An attempted solution is then notated in informal code, at a high level of abstraction, as if there were a machine that would be capable of interpreting that code and executing it. Then the refinement process begins. Step by step the solution is made more specific. At each new level of abstraction one takes new design decisions, but not more than strictly necessary. The design decisions are taken at such a level that they convey precisely the substructure to which they apply, not more and not less. The refinement of the program can go hand in hand with the closer inspection of the problem being solved:

> "As the problem analysis proceeds, so does the further refinement of

my program."
(Dijkstra, in: Dahl, Dijkstra & Hoare '72, pg. 27).

Clearly, this is an idealized description of a design process. The programmer is bound to make mistakes. He will discover in refinement-step x that he made a mistake in refinement-step x-1, or earlier. One then returns to the point where the mistake was made and continues with a different approach. The design process is thus essentially an iterative process, though in its idealized form one is tempted to define it as a one-pass procedure.

Each completed level of abstraction can represent an entire class of problems. In the process of refinement the class of problems is, however, narrowed down to precisely the problem that is to be solved by the program desired. If one day the requirements are reformulated, one merely returns to the appropriate level of abstraction and refines the solution anew, to satisfy the new requirements. Dijkstra writes:

"... when a program has been built up to an intermediate stage of refinement, what has then been written down is in fact a suitable 'common ancestor' for all possible programs produced by further refinements."
(Dijkstra, in: Dahl, Dijkstra & Hoare '72, pg. 40).

Together with the refinement of the program one refines the structure of the data. Decisions about the implementation of high level data structures are postponed for as long as can be.
The high level, or pseudo-code (the latter term is from **Melekian '76**) can be regarded as a form of 'structured comment' to the code of the levels immediately below it in the design hierarchy. The design process naturally stops when then lowest level has been reached, that is a description in the code of the programming language used. Gries, therefore, aptly remarked that with structured programming:

"... one doesn't program *in* a programming language but *into* it."
(Gries '74, pg. 656).

3.2.4. Notation

We discuss four items which are related to a clear notation of problems and solutions: data representations, flow-charts, documentation, and programming languages.

Data representations
Many articles have been published on the question of what type of data representation would lend itself best to the structured programming techniques (for instance, **Hoare**, in: **Dahl, Dijkstra & Hoare '72; Shneiderman & Scheuerman '74**; see also **Jackson '75**).
Two important articles have not gained the attention they deserve. One of them was published just before the wide discussions on programming style began (1967), the other a few years later (1971). We consider these two articles briefly below.

The first article is **Balzer '67**. Balzer proposed to keep the data manipulations in the design phase independent of specific data representations. The programmer then uses informal operations in his program text, like 'delete record', or 'replace', leaving the refinements to a later stage in program design. The method fits well in the stepwise refinement procedures outlined earlier. We quote:

"... programming should be simplified because it can be constructed top-down in terms of the logical processing required. The problem

> of data representation can be left until this programming has been completed; thus, a more rational decision can be made concerning an optimal representation. Because of this separation, the programmer should be able to think through his problem better."
> **(Balzer '67, pg. 542).**

Balzer suggested using an informal hierarchical data reference method, in which data can be accessed at the desired level of abstraction by statements of the type:

$$\text{name}_{\text{level n}}(\text{index}) \quad of \quad \text{name}_{\text{level n-1}}(\text{index}) \quad of \quad \ldots$$

The level of formal parameters need not match the level of the corresponding actual parameters (see also **Ross** *et al.* **'75.**)

The second article is by **Early '71.** Early described the 'V-graph model' for data representations. These V-graphs represent the set of data on an arbitrary level of abstraction, and not (necessarily) the way in which the data is implemented **(Early '71, pg. 618).**
An example of a V-graph is given in figure 3.3. Early explained the meaning as follows:

> "Each node of the graph represents a part of the data structure. The arrows between nodes represent access paths in the structure, and arrows from nodes to 'atoms' represent the fact that the atom is stored at the node."
>
> "NIL is a reserved object which can be used for such things as the end of a list."

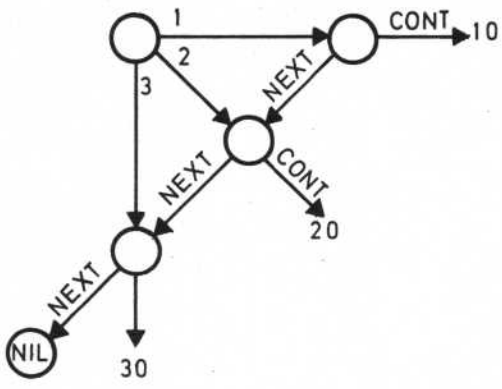


Figure 3.3.

Early's V-graph model

The annotations on the edges of the graph indicate subscript variables (or 'selectors' as Early calls them). In the example of figure 3.3 the elements 10, 20 and 30 can be accessed by subscripts as well as via the standard operation NEXT. The names NEXT and CONT are not reserved words though; one can use any other suitable notation (PRIOR, FOLLOW, ADJACENT, IMAGPART, REALPART, AGE, YEAR, etc.).

The importance of a clear notation is easily underestimated. An obtuse notation

however can make a problem practically unsolvable. An adequate data representation can facilitate the solution of the related problem in many ways. These ideas have been stressed by many, among others by: **Polya '48**, and later by **Hoare** (in: **Dahl, Dijkstra & Hoare '72**), and **Baker '75**.
The latter especially stressed the importance of 'intelligent data names'.

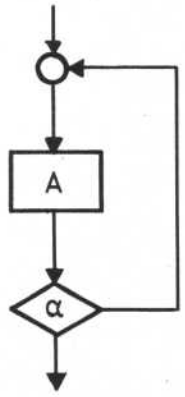
Flow-charts
Judged by the standards of structured programming, the conventional flow-charting techniques fall short on three major points:

(1) There is no clear representation of the three basic control-flow structures; the representation of iteration structures is inadequate.
(2) The programmer is in no way restricted in his use of arbitrary GO TO jumps, on the contrary.
(3) There is no clear representation of hierarchical levels in a program.

For these reasons one has found that the use of conventional flow-charts, especially in the design stages, conflicts with the aims of structured programming. Several attempts have been made to improve the flow-charting technique. One of the additional rules that have been suggested is:

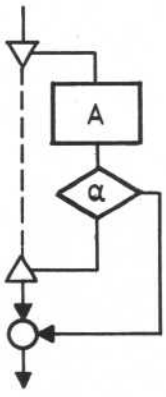
- all jumps (including the iteration jumps) must be drawn on one side of the flow-chart, while the
- crossing of lines in the flow-chart is forbidden.

This rule would compensate for the second shortcoming mentioned above, but leaves the first and the third.
Two of the proposals for alternative representations of iteration constructs in flow-charts are given in figure 3.4. Neither has gained much support. Observe that the third objection against the traditional flow-charts is still valid.



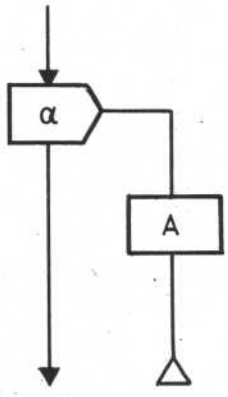
(a)

Conventional Flowchart-representation of Iteration.



(b)

Alternative Suggestion of **Schmitz '75.**



(c)

Alternative Suggestion of **Weiderman** *et al.* **'75.**

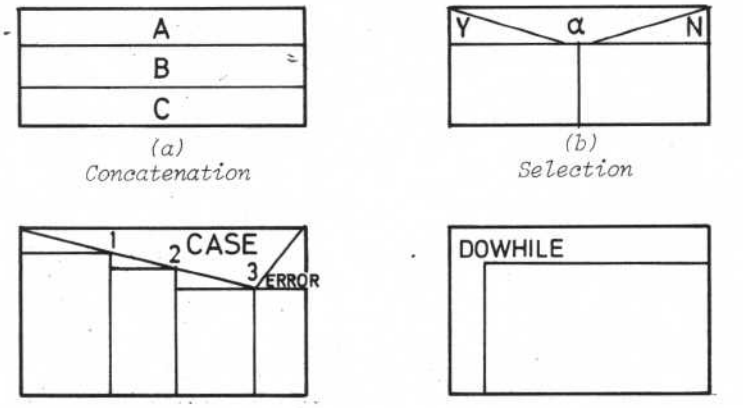
In 1973 Nassi & Shneiderman suggested quite another approach to the problem, and their attempt was more successful. Nassi & Shneiderman introduced a new type of flow-chart which removes all objections mentioned, to a large extent. In addition, they noted that:

"... the scope of local and global variables is immediately obvious."

"... complete thought structures can and should fit on no more than one page (i.e. no off-page connectors)."

"... recursion has a trivial representation."
(Nassi & Shneiderman '73, pg. 15).

The three basic control-flow-structures are represented in the new model as indicated in figure 3.5. The representation of a block or hierarchical level is indicated in figure 3.6. The representation of a recursive subroutine is illustrated in figure 3.7.



(a) Concatenation

(b) Selection

(c) Case Selection

(d) Iteration

Figure 3.5. Structogramme Notations

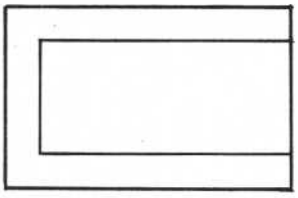


Figure 3.6. Block or Level

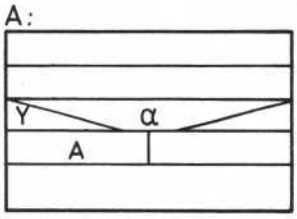


Figure 3.7. Recursive Subroutine

Loop-exit or error-exit jumps are more difficult to represent, as could be expected. Nassi & Shneiderman suggested using the key-word BREAK in such cases. The type of flow-charting suggested by Nassi & Shneiderman is called 'structured flow-charting' or the 'structogramme notation'. The term 'structogramme' stems from **Schnupp '75** (see further **Moos & Steinbuch '76**).

Still, another approach to the problem of representing a program structure was suggested by **Pazel '75.** Pazel suggested a Boolean-like notation for the three basic structures.
The concatenation of three statements a, b, and c is then indicated as a 'dot-product': a.b.c .
A selection between statements a and b is indicated as: $(a +_{\beta} b)$, where β is the 'selection criterion'.
An iteration on statement a with iteration predicate α is indicated with so-

called 'jump-indicators' (J) and 'receptor-indicators' (R): R.a $(1 + {}_{\alpha}J)$. The latter notation is rather inelegant, and could perhaps be replaced by a notation of the type (a^{α}), while avoiding the J- and R-indicators.

Unfortunately, the resemblence with a Boolean algebra is not valid for the manipulations of the 'formula's'. For instance, it can be shown that the structure which corresponds to the formula: $(a + \beta b).c$ is equivalent to the structure corresponding to the formula: $(a.c + {}_{\beta}b.c)$. In other words: it can be shown that the dot-operation is 'right-distributive' over the +. To show that the structure given in figure 3.8.a is equivalent to that of figure 3.8.b, or that the relation:

$$R_1\ (R_2.a + {}_{\alpha}J_1).b.(1 + {}_{\beta}J_2) \triangleq R_2.R_1\ (S_1 + {}_{\alpha}J_1).b.(a + {}_{\beta}J_2)$$

holds, is however much more complicated.

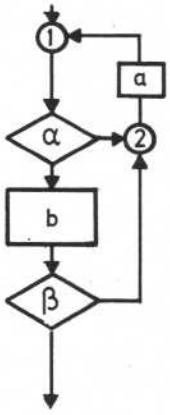


Figure 3.8.(a)
Pazel Notation:
$R_1.(R_2.a +_{\alpha} J_1).b.(1 +_{\beta} J_2)$

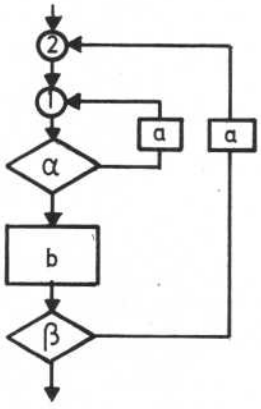


Figure 3.8.(b)
Pazel Notation:
$R_2.R_1.(S_1 +_{\alpha} J_1).b.(a +_{\beta} J_2)$

The approach suggested by Pazel deserves more attention then can be paid to it in the present context. We leave this as a recommendation for further research.

Documentation
We have noted that the abstract solutions which are drafted in the process of stepwise refinement can serve as structured comments to the derived program. Linger & Mills have suggested using still more stringent rules for the inclusion of comments in a program text. They have stated that all data structures and all actions taken in a program should be explained explicitly in the form of comments to the code.

> "In general, every data object introduced should have a data definition giving its contextual meaning. An action definition should precede the program part it refers to, and a status definition should follow the program part which produced the status."
> **(Linger & Mills '75, pg. 179).**

Linger & Mills further suggested preceding each iteration loop with a specification of the corresponding loop-invariant assertion.

In **Bulow '74** it is suggested to concentrate all comments to a program text in a sort of 'table of contents' at the beginning of the program. The only com-

ments which appear in the program text itself are then the 'chapter headings' etc. Similar proposals were published by **Berry '75**.

One other important method of making a program text more understandable is the *hierarchical indentation* of that text. Clearly, this indentation is merely an additional tool to make structured programs understandable. It will, however, not make a poor design much stronger. An amusing example of this was given by **Van Amstel '77**. Van Amstel quoted a person who had the impression that structured programming boiled down to requiring that sentences like:

> 'The one who denounces the one who pulled down the pole that stood near the bridge in the road that leads to Worms which was recently opened over the canal of ten feet tall within ten days will be rewarded.'

should be written as follows:

```
'The one
 who denounces the one
  who pulled down the pole
   that stood near the bridge
    in the road
     that leads to Worms
     which was recently opened
    over the canal
   of ten feet tall
  within ten days
 will be rewarded.'
```

A well-structured 'sentence', however, should have quite another structure. In Van Amstel's opinion the well-structured equivalent of the sentence given above would be **more like:**

> 'In the road that leads to Worms there is a bridge over a canal. This bridge was recently opened. Near that bridge there stood a pole of ten feet tall. This pole has been pulled down. The one who reports the responsible person within ten days, will be rewarded.'

(The original Dutch example is more concise, but it could not be translated literally without losing the characteristic structure.)

Programming languages

The question of whether the existing programming languages are suited to structured programming practice has been raised repeatedly. Already in 1962 McCarthy wrote:

> "At present, programming languages are constructed in a very unsystematic way. (-) A better understanding of the structure of computations and of data spaces will make it easier to see what features are really desirable."
> **(McCarthy '62).**

More than once the language FORTRAN has been criticized for its lack of structure, and the omnipresence of goto-jumps. But, FORTRAN is not really an exception: in fact no language seems to meet the standards completely.
Lecarme posed the rhetorical question to his readers:

> "Is it possible to write structured programs (-) in any language?"

Lecarme answered promptly:

> "The answer is certainly no, and this has already been said for several times."
> **(Lecarme '74, pg. 16).**

The subject was discussed among others by **Wulf** *et al.* **'71; Peck '72; Clark** *et al.* **'73; Wirth '74,** and in **Update, Vol. 2, No. 5, pg. 8-10, May 1977.** In **Lecarme '72** the following requirement was formulated:

> "... conciseness, structure and clarity (-) are probably the most important qualities of any program. We intentionally omit efficiency, because we believe that a program that is concise, structured and clear must be efficient otherwise the programming language is bad."

The programming languages which seem to answer this requirement best include ALGOL-68, ALGOL-W, and PASCAL.

3.3. DISCUSSION

The structured programming methodologies have been critized quite often. Jones noted that:

> "There is often, especially initially, some consumer resistance to the idea when it is first preached to practising programmers. (-) But, after some oscillations between 'it's nonsense' and 'we've always done that' most programmers would be prepared to use such a framework for recording their work."
> **(Jones '77, pg. 315).**

Gries gave a plausible explanation for this initial resistance:

> "The defensive attitude about structured programming (-) occurs partly because people object to being told that they don't know how to do their jobs efficiently."
> **(Gries '74, pg. 656).**

More fundamental objections have however been raised against the methodology, which we will consider here. The criticisms can be divided into four topics:

- pragmatics,
- complexity,
- rigidity, and
- methodology.

3.3.1. Pragmatics

A very practical objection raised against the structured programming techniques is that many programming languages lack the adequate facilities for it **(Abrahams '75).** The most widely used programming languages (FORTRAN, COBOL, PL/1, ALGOL 60) are not really suited for the techniques. They even tend to make the structured programs less efficient. The rebuttal to this objection can clearly be that if no adequate languages are available at the moment, they should be made available on short notice. Lately, several attempts have been made to define more suitable languages (ALGOL-W, PASCAL, etc.). The chance that they will eventually repress the existing languages is however not always estimated as high.

3.3.2. Complexity

Another criticism is that the structured design techniques are in fact just as complicated as the traditional methods **(Abrahams '75).** To choose the right data representation, the right program structure, is still far from trivial. It is then argued that 'poor programmers' will still write 'poor programs' even if they apply the structured programming techniques. This criticism can however be refuted. Structured programming cannot pretend to be a 'magic wand' which alters 'poor programmers' into first-class programmers. It does, however, offer the instruments for good program design. As always these instruments can be used correctly and incorrectly. Denning explained:

> "The understandability of the product remains basically a matter of style: some programmers have good style, others do not. Good programming style is not automatically introduced by the rules of structured programming, any more than good English prose style is guaranteed by following the ten famous rules listed in Strunk & White's: *The Elements of Style* ... though these rules can go a long way toward influencing programmers or writers in the direction of good style, ..."
> **(Denning '74, pg. 6).**

3.3.3. Rigidity

Many authors have 'complained' that the rules of structured programming are too strict. Some say that their strict application can lead to inferior programs. Some even claim that their strict application is impossible.

Abrahams, for instance, questions whether each program can be given a 'tree-structure' (hierarchical structure), whether global variables can always be avoided in large programs, and whether goto-jumps can always be avoided. Johnson stated much earlier:

> "In some cases, it is difficult or impossible to write a program without crossing one branch line by another."
> **(Johnson '70, pg. 122).**

Berzheim '75 further objected that:

(1) the iteration construct is sometimes too strong, for instance, for program segments that must be repeated just once;
(2) overlapping iteration structures can be of value, but are forbidden;
(3) there is a need for the representation of 'error-exits'.

The need for a clear notation of 'error-exits' has been stressed repeatedly, and as such it would of course not contradict the goals of structured programming to include such a special construct in a language. The flow-of-control in a program should however be prevented from being diffused by such special constructs. One obvious requirement is then that it should not be possible to 'jump' from the error-routines back into the program at some intermediate point.

Another item in what has been called the 'goto-controversy' is the use of loop-exit structures. The classic example is the 'search-loop'. A search-loop has two or more possible outcomes: the search is successful (in one or more ways) or it is not. Each outcome may require a different response. Without special constructs for the programming of such loop-exit structures one is tempted to use goto-statements instead.
The represenation of the loop-exit structures is greatly facilitated if one decouples the declaration of iteration conditions and loop-structures.

In the traditional loop constructs WHILE(...)DO ... OD and REPEAT ... UNTIL(..) the iteration condition must be evaluated either at the start or at the end of the loop-body; they can however not be evaluated within the loop-body.
An alternative construct, much in line with the proposals of e.g. **Knuth & Floyd '71**, and **Wirth '67 (pg. 29)** would be to use the key-word REPEAT in combination with the exit-statements EXITIF(condition; DOING: statements). The exit should then be from the innermost block or loop. With a still more liberal approach we could allow for exits from any level in the nesting hierarchy. The level to be exited must then be specified: EXITIF(condition; FROM: level; DOING : statements).
In figure 3.9 we have given a complex non-decomposable flow-chart discussed by **Peterson** *et al.* **'74.** In figure 3.10 the description of that flow-chart in a structogramme, extended with EXITIF statements, is given. The structogramme was obtained by a redrawing of the flow-chart into the more structured one of figure 3.11 in which a few blocks of code have been duplicated. For comparison we reproduce the notation of the same flow-chart with the aid of a WHILE-UNTIL construct (without code-duplications), taken from **Friedman & Shapiro '74 (pg. 11)**, below.
The difference in clarity of the control flow in the programs need hardly be stressed.

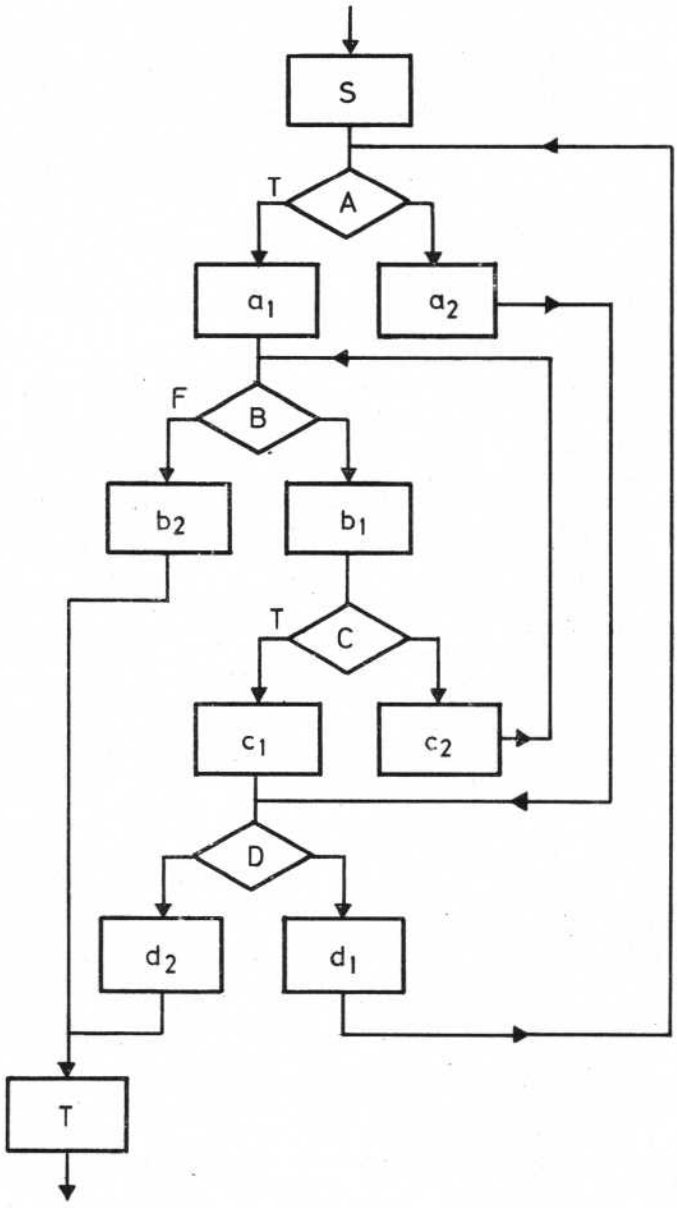


Figure 3.9.

Non-Decomposable Flow Chart

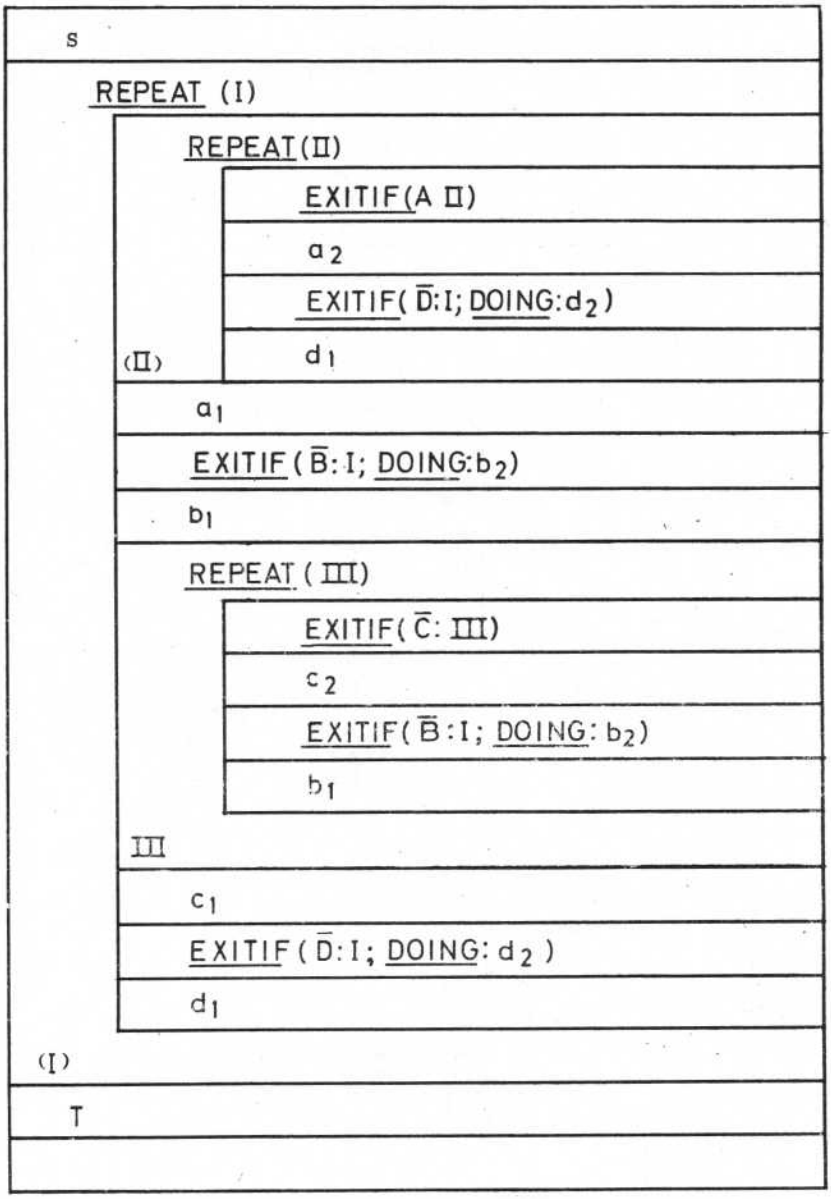


Figure 3.10.
Structogramme with REPEAT/EXITIF Construct

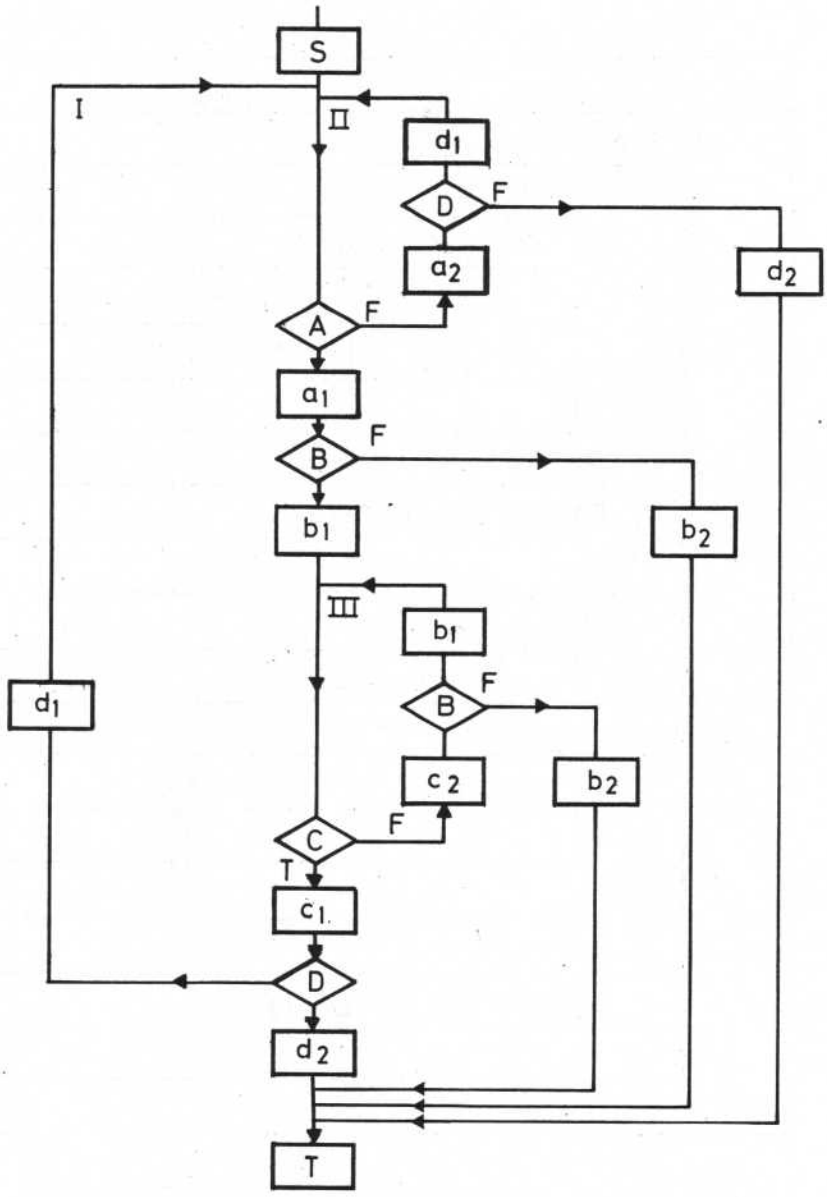


Figure 3.11.
Redrawn Flowchart for Fig. 3.9.
(with some code-duplications)

```
S; while (if A
            then (if (while true repeat a1 until true)
                    then (if (while B repeat b1 until
                                (if C then true
                                   else not(while true repeat c2 until true)))
                             then (while true repeat c1 until true)
                             else not(while true repeat b2 until true))
                    else false)
            else (while true repeat a2 until true))
   repeat until (if D then not(while true repeat d1 until true)
                   else (while true repeat d2 until true)); T
```

3.3.4. Methodology

The most fundamental criticisms on structured programming refer to the actual design process and the relation to problem solving techniques.
In the first place there is still confusion about the true nature of the design process. The question is whether structured program designing implies pure 'top-down' programming or not.
Denning says:

> "I am not sure what 'top-down programming' means; I have seen the phrase used in at least two ways:
>
> a. It is a method of decomposition into modules by stepwise refinement.
> b. It is the goal of evolving a program so that its final structure can be presented or described as a hierarchy of tasks or blocks."
> **(Denning '74a, pg. 7).**

Many authors identify structured programming with top-down programming in the first sense (a). Peter Naur, for instance, wrote:

> "Thus, as one example, the insistence of Dijkstra and Wirth on stepwise refinement could well get in conflict with the first two problem solving precepts of Hyman and Anderson **(Hyman & Anderson '65)**:
>
> Precept I: 'Run over the elements of the problem in rapid succession several times, until a pattern emerges which encompasses all these elements simultaneously';
> Precept II: 'Suspend judgement. Don't jump to conclusions.' "
> **(Naur '72, pg. 348).**

The conflict described by Naur is however only apparent. The first precept Naur gives resembles Polya's advice to 'discover the major lines of the argument' and the rule of abstraction from structured programming.
The second precept is completely in agreement with the design rule of stepwise refinement, and confirms Polya's remark:

> "... it is not reasonable to check minor details before we have good reasons to believe that the major steps of the argument are sound."
> **(Polya '48, pg. 69).**

Every programmer knows from his own experience that the task of designing program, or the task of solving a problem, is hardly ever executed as a pure top-down process. Denning (citing Wilkes) made the following illustrative comparison:

> "[**compare**] writing a textbook to writing a program: (-) the process of creating, and of evolving, that structure can hardly be charac-

> terized as a top-down process (-) it is a matter of individual creativity and style."
> **(Denning '74b, pg. 6).**

That program designing or problem solving is no pure top-down process does however not imply that it lacks all structure. Oddly enough Polya gives his readers that impression when he gives the following, playful description of the 'rules of discovery':

> "The first rule of discovery is to have brains and good luck.
> The second rule of discovery is to sit tight and wait till you get a bright idea."
> **(Polya '48, pg. 172).**

Some problems are however so inherently complex that indeed no method other than the ones described here by Polya may be of use. One such a problem is the problem of the 'Tower of Hanoi'. **Peck '72** made a comparison of the abilities of a number of programming languages to give a clear representation of the program which solves this problem. None of the examples is very understandable though. These observations do, however, not really undermine the applicability of the structured programming techniques, especially because there are no alternative techniques which have proved to be more powerful.

3.4. CONCLUSION

The four principles of structured programming (abstraction, restriction, stepwise refinement, and clear notation) are powerful instruments for the discovery of transparent solutions of programming problems. The principles can be used as a guideline in program design; they should however not be interpreted as 'dogma's'. The principles are intended as a *means* to realize the specific goal of understandability; they are not an independent goal in themselves.

3.5. REFERENCES

Abrahams, P. (1975), *Structured programming considered harmful*, Sigplan Notices, Vol. 10, No. 4, pp. 13-24, April 1975.

Ashcroft, E. & Manna, Z. (1971), *The translation of 'goto' programs into 'while' programs*, Computer Science Dept., Report CS-188, Stanford University, also presented at IFIP '71, Jan. 1971.

Baker, F.T. (1975), *The pros and cons of structured programming*, Data Management, Vol. 13, No. 9, pp. 60-71, Sept. 1975.

Balzer, R.M. (1967), *Dataless programming*, Proc. AFIPS 1967 FJCC, AFIPS Press, Montvale (N.J.), pp. 535-544, 1967.

Berry, D.M. (1975), *Structured documentation*, Sigplan Notices, Vol. 10, No. 11, pp. 7-12, Nov. 1975.

Berzheim, H. (1975), *Fernwirkungsfreie, strukturierte Programmierung*, Online, Vol. 13, No. 4, pp. 234, 237-240, 1975.

Bloom, A.M. (1975), *The 'else' must go, too*, Datamation (U.S.A.), Vol. 21, No. 5, pp. 123-128.

Bohm, C. & Jacopini, G. (1966), *Flow diagrams, Turing machines and languages with only two formation rules*, Comm. ACM, Vol. 9, No. 5, pp. 366-371, May 1966.

Bulow, K. (1974), *Programming in book format*, Datamation, Vol. 20, No. 10, Oct. 1974.

Canning, R.G. (ed.) (1974), *The advent of structured programming*, EDP Analyzer (U.S.A.), Vol. 12, No. 6, pp. 1-14, June 1974.

Clark, B.L. *et al.* **(1973)**, *Reflections on a language design to write an operating system*, Sigplan Notices, Vol. 8, No. 9, 1973.

Cooper, D.C. (1967), *Böhm and Jacopini's reduction of flow charts*, Comm. ACM, Vol. 10, No. 8, pp. 463, 1967.

Dahl, O.J., Dijkstra, E.W. & Hoare, C.A.R. (1972), *Structured programming*, Academic Press, London, 1972, 220 pgs.

Davies, I.K. (1971), *The management of learning*, McGraw-Hill, London, 1971.

Davies, I.K. (1976), *Objectives in curriculum design*, McGraw-Hill, London, 1976.

Denning, P.J. (1974a), *Is it not time to define structured programming?*, Operating Systems Review, Vol. 8, No. 1, pp. 6-7, Jan. 1974.

Denning, P.J. (1974b), *Is structured programming any longer the right term?*, Operating Systems Review, Vol. 8, No. 4, pp. 4-6, April 1974.

Dennis, J.B. (1975), *Modularity*, Lect. Notes in Computer Science, Vol. 30, pp. 128-182, Springer Verlag, 1975.

Donaldson, J.R. (1973), *Structured programming*, Datamation, Vol. 19, No. 12, pp. 52-54, 1973.

Dijkstra, E.W. (1961), *On the design of machine independent programming languages*, Report MR34-Math. Centre, Amsterdam, Oct. 1961, 18 pgs.

Dijkstra, E.W. (1968a), *GO TO statement considered harmful*, Comm. ACM, Vol. 11, No. 3, pp. 147-148, 538, 541, March 1968.

Dijkstra, E.W. (1968b), *The structure of the 'THE' multiprogramming system*, Comm. ACM, Vol. 11, No. 5, pp. 341-346, May 1968.

Dijkstra, E.W. (1969a), *Complexity controlled by hierarchical ordering of function and variability*, In: **Naur, P. & Randell, B. (eds.)**, *'Software Engineering'*, Sc. Aff. Div., NATO, Brussels, pp. 114-116, 1969.

Dijkstra, E.W. (1969b), *Notes on structured programming*, THE Report, Univ. of Techn. Eindhoven, The Netherlands, EWD-249, 70-Wsk-03, August 1969, see also **Dahl** *et al.* **'72**.

Dijkstra, E.W. (1972), *Hierarchical ordering of sequential processes*, Operating Systems Techniques, New York, Academic Press, 1972.

Dijkstra, E.W. (1975), *Correctness concerns and, among other things, why they are resented*, Sigplan Notices, Vol. 10, No. 6, pp. 546-550, 1975.

Dijkstra, E.W. (1976), *A discipline of programming*, Englewood Cliffs, Prentice Hall, 1976, 220 pgs.

Early, J. (1971), *Toward an understanding of data structures*, Comm. ACM, Vol. 14, No. 10, pp. 617-627, Oct. 1971.

Evans, R.V. (1974), *Multiple exits from loops using neither goto nor labels*, Comm. ACM, Vol. 17, No. 11, pg. 650, Nov. 1974.

Flon, L. (1975), *On research in structured programming*, Sigplan Notices, Vol. 10, No. 10, pp. 16-17, 1975.

Friedman, D.P. & Shapiro, S.C. (1974), *A case for 'while-until'*, Sigplan Notices, Vol. 9, No. 7, pp. 7-14, July 1974.

Garren, S. (1973), *Structured programming*, DECUS Fall 1973, Symposium, San Fransisco, Cal., U.S.A., Nov. 1973, pp. 37-39.

Goodenough, J.B. (1975), *Structured exception handling*, In: Conference Record of the 2nd ACM Symposium on the *'Principles of programming languages'*, pp. 204-224, Jan. 1975.

Gries, D. (1974), *On structured programming, a reply to Smoliar*, Comm. ACM, Vol. 17, No. 11, pp. 655-657, Nov. 1974.

Harding, D.A. (1972), *Modular programming - why not!*, Austr. Comput. J., Vol. 4, No. 4, pp. 150-156, Nov. 1972.

Hyman, R. & Anderson, B. (1965), *Solving problems*, Int. Science and Technology, Sept. 1965, pp. 36-41.

Infotech (1978), *Structured programming, practice and experience*, Infotech State of the Art Report, 1978, 900 pgs., 5 Volumes, Infotech Int. Ltd., Berkshire, U.K.

Jackson, M.A. (1975), *Principles of program design*, AFIC Studies in Data Processing, No. 12, Hoare, C.A.R. (ed.), Academic Press, 1975.

Johnson, L.R. (1970), *System structure in data, programs, and computers*, Prentice Hall, Englewood Cliffs (N.J.), 1970.

Jones, C.B. (1977), *Structured design and coding: theory versus practice*, Informatie, Vol. 19, No. 6, pp. 311-319, June 1977.

Kernighan, B.W. & Plauger, P.J. (1974), *Programming style: examples and counter examples*, Computing Surveys, Vol. 6, No. 4, pp. 303-319, 1974.

Knuth, D.E. & Floyd, R.W. (1971), *Notes on avoiding goto statements*, Inf. Proc. Letters, pp. 23-31, 1971.

Knuth, D.E. (1974), *Structured programming with 'goto' statements*, Computing Surveys, Vol. 6, No. 4, pp. 261-301, 1974.

Kuhn, T. (1962), *The structure of scientific revolutions*, Univ. of Chicago Press, Chicago & London, 1962.

Leavenworth, B.M. (ed.) (1972), *The 'goto' controversy, a debate, (Programming with(out) the 'goto'*) Proc. 1972 ACM, National Conference, Vol. 2, pp. 782-797.

Lecarme, O. (1973), *What programming language?*, Proc. IFIP-TC-2, Working Conf. on *'Programming Teaching Techniques'*, Poland, Sept. 1972, North-Holland Publ. Co., Amsterdam 1973, pp. 61-74.

Lecarme, O. (1974), *Structured programming, programming teaching and the language PASCAL*, Sigplan Notices, Vol. 9, No. 7, pp. 15-21, July 1974.

Linger, R.C. & Mills, H.D. (1975), *Definitional text in structured programming*, Proc. of the 14th Annual Technical Symposium on: *'Computing in the mid 70's: and assesment'*, Gaithersburg (Md.), June 1975. Reprinted in: Proc. IEEE Compcon '75, *Tutorial on Structured Programming*, pp. 178-183.

Liskov, B.H. (1972), *A design methodology for reliable software systems*, Proc. of the 1972 FJCC, AFIPS Press, Montvale (N.J.), pp. 191-199, 1972.

McCarthy, J. (1963), *Towards a mathematical science of computation*, Proc. IFIP Congress 1962, Münich, North-Holland Publ. Co., Amsterdam, pp. 21-28, 1963.

McClure, C.L. (1975), *Top-down, bottom-up, and structured programming*, IEEE Trans. on Software Eng., Vol. SE-1, No. 4, pp. 397-403, Dec. 1975.

McCracken, D. (1973), *Revolution in programming, an overview*, Datamation, Vol. 19, No. 12, pp. 50-52, Dec. 1973.

Melekian, N. (1976), *New methods and techniques of programming*, (in German), IBM Nachrichten, Vol. 26, No. 299, pp. 48-55, 1976.

Mills, H.D. (1972), *Mathematical foundations of structured programming*, IBM, Federal Systems Div., Gaithersburg, Maryland, 1972.

Moos, A. & Steinbuch, P.A. (1976), *Structogramme and structured programming*, (in German), Buerotechnik, Vol. 24, No. 9, pp. 39-43, Sept. 1976.

Nassi, J. & Shneiderman, B. (1973), *Flow chart techniques for structured programming*, Sigplan Notices, Vol. 8, No. 8, pp. 12-26, 1973.

Naur, P. (1966), *Proof of algorithms by general snapshots*, BIT, Vol. 6, pp. 310-316, 1966.

Naur, P. (1972), *An experiment on program development*, BIT, Vol. 12, pp. 347-365, 1972.

Newton, G.E. (1975), *A partially annotated bibliography of top-down and gotoless programming*, In: Proc. *'Proving and improving programs'*, Arc et Senans, France, July 1975, Rocquencourt, IRIA, pp. 435-473, 1975.

Parnas, D.L. (1972), *On the criteria to be used in decomposing systems into modules*, Comm. ACM, Vol. 15, No. 12, pp. 1053-1058, Dec. 1972.

Pazel, D.P. (1975), *Mathematical construct for program reorganization*, IBM Journal of Research & Development, Vol. 19, No. 11, pp. 575-581, Nov. 1975.

Peck, J.E.L. (1973), *Comparison of languages*, In: *Programming Teaching Techniques*, Proc. IFIP 1972, North Holland Publ. Co., Amsterdam 1973.

Peterson, W.W., Kasami, T. & Tokura, N. (1973), *On the capabilities of 'while', 'repeat', and 'exit' statements*, Comm. ACM, Vol. 16, No. 8, pp. 503-512, '73.

Polya, G. (1971), *How to solve it*, Doubleday Anchor, New York, 1948 and 1957, Reprinted by: Princeton Press, 1971.

Ross, D.T. *et al.* (1975), *Software engineering, process, principles and goals*, Computer, Vol. 8, pp. 17-27, May 1975.

Schmitz, A. (1975), *Das strukturierte Ablaufdiagram zur graphischen Interpretation von Programm-Abläufen*, (in German), Online, Vol. 13, No. 10, pp. 665-667, 1975.

Schnupp, P. (1974), *Structogramme - a new method of system planning*, (in German) Online, Vol. 12, No. 11, pp. 736-743, 1974.

Shneiderman, B. & Scheuerman, P. (1974), *Structured data structures*, Comm. ACM, Vol. 17, No. 10, pp. 566-574, 1974.

Van Amstel, J.J. (1977), *Methods for structured programming*, (in Dutch), Informatie, Vol. 19, No. 6, pp. 325-338, June 1977.

VanderBrug, G.J. (1974), *On structured programming and problem reduction*, Report NSF-OCA-GJ32258X-29, Univ. of Maryland, College Park, U.S.A., Jan. 1974, 11 pgs.

Weiderman, N.H. *et al.* (1975), *Flow-charting loops without cycles*, Sigplan Notices, Vol. 10, No. 4, pp. 37-46, 1975.

Weinberg, G.M. (1971), *The psychology of computer programming*, New York, Van Nostrand Reinhold Co., 1971.

Weinberg, G.M., Geller, D.P. & Plum, T.W.S. (1975), *'If-then-else' considered harmful*, Sigplan Notices, Vol. 10, No. 8, pp. 34-44, August 1975.

Wirth, N. (1967), *On certain basic concepts of programming languages*, Report Stanford Univ.,Comp. Sc. Dept., May 1967, 30 pgs.

Wirth, N. (1971), *Program development by stepwise refinement*, Comm. ACM, Vol. 14, No. 4, pp. 221-227, April 1971.

Wirth, N. (1974), *On the composition of well structured programs*, Computing Surveys, Vol. 6, No. 4, pp. 247-259, 1974.

Wulf, W.A. *et al.* (1971), *Reflections on a systems programming language*, Sigplan Notices, Vol. 6, No. 9, 1971.

Wulf, W.A. & Shaw, M. (1973), *Global variables considered harmful*, Sigplan Notices, Vol. 8, No. 2, pp. 28-34, Febr. 1973.