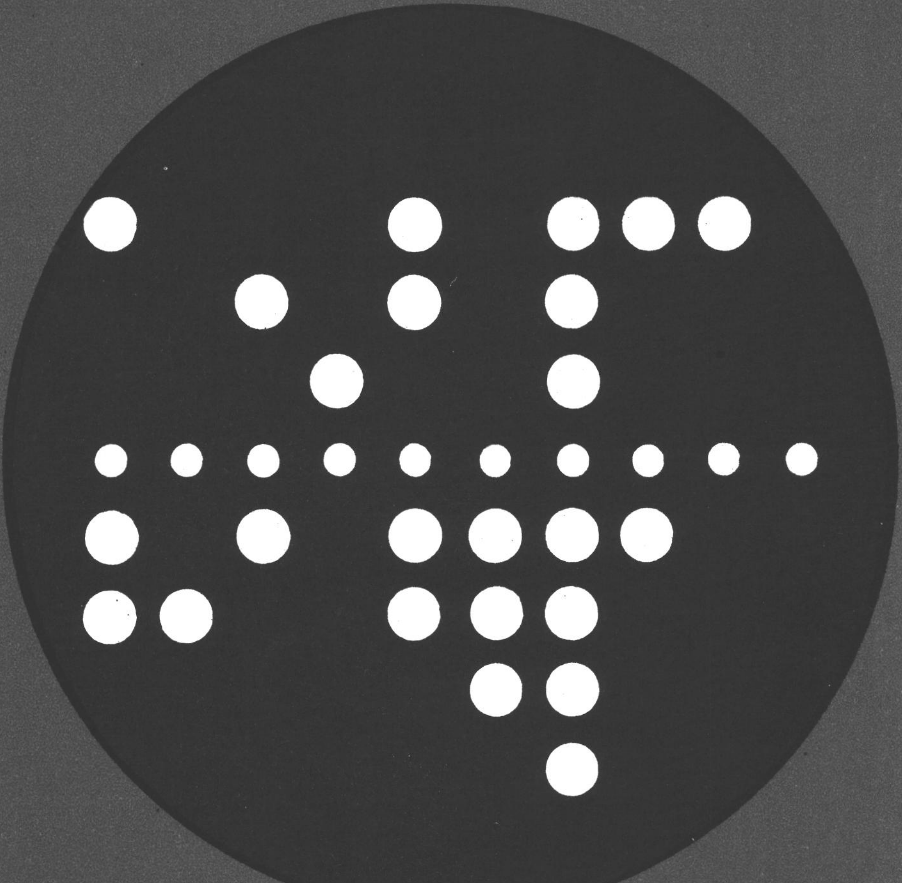
COMPUTING CENTRE NEWSLETTER

TOWARDS A NEW PROGRAMMING STYLE



Commission of the European Communities

JOINT RESEARCH CENTRE

Ispra Establishment

SPECIAL ISSUE

TOWARDS A NEW PROGRAMMING STYLE

A.A. Pollicini

June 1978

CONTENTS

INTRODUCTION

What, in fact, is Programming?
Activities carried out during the past years under the name "programming" can be considered to be the first phase of man-computer communication; that is, the phase in which man speaks and the computer listens.

As Donald E. Knuth has pointed out, programming can be considered to be an art [1].

Dante and Shakespeare were artists in transmitting thoughts and feelings expressed by language and they are admired for this ability. However, generations of people are still using the same languages to communicate daily with each other and it is only when the language rules are observed that the resulting expressions are correct and pleasant and above all that understanding is unequivocal.

It is therefore sensible to promote and demand correctness in all, even humble, language applications.
This rule also applies to programming languages.

Moreover, programming languages are recent tools, still in an evolutionary phase. The use of these languages can therefore be improved , by improving the languages themselves.
The more the rules of the languages are logically defined and rigorously applied the more the resulting expressions become correct and intelligible.

The definition of grammar, syntax and semantics of programming languages can profit considerably from an understanding of the mechanisms of human intelligence which also inspired the binary logic on which the computer itself is based [2].

REASONS FOR NEW PROGRAMMING TECHNIQUES

Programming Evolution

When automatic computing machines first appeared, users found themselves faced with a largely unknown tool, the computer, waiting for commands to be executed.
The collection of instructions needed to communicate its tasks to it was called a "Program".

In those early computer applications, a programmer, like a sort of modern alchemist, might oversee effects rather than causes. Indeed, he manipulated the data controlling the results, while remaining unaware of the processes which generated them.
At that time, the first property a Program had to have, was synthesized by the phrase "provided that it works".

The cost of this "empirical programming" was heavy maintenance due to the difficulties in extending or modifying the programs and limiting their lifetime. Briefly a considerable waste of resources.

Gradually, however, a number of research activities were begun in order to develop a better understanding of what Programming really is.

From a the functional point of view a Program is a sequence of actions, each of them accomplishing a function. At this point a correct and unequivocal definition of each function appears as a pre-requisite that cannot be ignored.
Moreover the effect of such a function is the transformation of some data. If there is any way to make axiomatic assertions concerning the expected transformation, it becomes possible to verify whether or not a Program performs the intended function [3].
By this means, searchers aim for a priori proof of Program correctness.

From the formal point of view a Program is a series of symbols reproducing in a one-dimensional manner the connected steps of the two-dimensional image of an algorithm.
The reduction from the planar representation (flow-chart) to the linear one (program) necessitates the introduction of jumps within the series which may involve some loss of clarity in the algorithm logic.
Since the flow-chart is a directed graph, a formal representation of all its paths by mathematical equation may provide the linguistic means to build well-formed programs.
Regular expressions, as formulated by Kleene in 1956 [4], could be applied satisfactorily to the purposes of flow-chart formalization.

These Regular Expressions connect some primitives using only three operators:

star	which means 'any number of times' represented by the symbol *
juxtaposition	which means 'followed by' represented by the absence of any symbol
alternation	which means 'or' represented by the symbol v

Their application to the analysis of programming represents the nodes of a flow-chart as primitives and the connections as the allowed operators.

Attempts to follow this approach showed that most of the complications a program can include were originated with the undisciplined branches that the common programming languages freely allow. It was this fact that led to the crusade against the GO TO [5]. If reduced to this question alone, the controversy would be a sterile one.
The real problem, however, was to provide a generalized method of expression, which could simulate any possible path in a nearly linear way.
An important theoretical contribution in this direction was made by the theorem of Böhm-Jacopini [6] which states:

Any flow diagram can be represented by a flow diagram that can be decomposed into concatenation, iteration and selection.

Of course, the practical impact of this on programming is somewhat weakened because, as the authors realized, the derived flow diagram is not always strongly equivalent to the original one.
Indeed, with the transformation of structures of the type Ω_n (i.e. loops with multiple exits) one notices a loss of clarity and effectiveness.
What is relevant is that the use of closed structures, delimited by one entry and one exit, however nested, can result in a descending flow of control within the Program, which assumes globally a linear pattern.
In programming practice more control structures are needed to represent all general situations effectively, but the principle of disciplined use of closed control structures remains valid.

The Intended Goal

The "Programming" action produces objects, usually called Program, System or Package according to the increase in size and complexity. But in common parlance all these items can be referred to by the single term "Software" which includes any codified knowledge to describe to the computer a task we want it to perform.
If there is any need to change the programming style, every innovation must yield an improvement in the Software production.

The criteria for the verification of new programming techniques depend upon one principle:

Good Techniques must generate Good Software

The evaluation of good Software depends on parameters that are the expected properties of the product.
Correctness is obviously the essential property in the sense that the Software must implicitly give a correct solution to the problem.
The other properties are differently weighted according to several aspects of Software production.
For instance, a classification of pre-eminent properties as a function of a considered aspect, might be:

- Effectiveness,Reliability	as a function of	exploitation
- Readability,Adaptability	as a function of	maintenance
- Portability,Adaptability	as a function of	distribution

Of course, some properties are contradictory, and compromises will therefore have to be made in order to achieve the best results, so that each Software package becomes the most satisfactory solution for its specific requirements.

THE ERA OF STRUCTURED PROGRAMMING

Structured Programming is the happy slogan of the 70's.
Everyone concerned with computer applications has at least heard this term.
A series of articles, workshops and courses have been promoting Structured Programming, nevertheless many people still identify the topic with a set of control structures, namely IF-THEN-ELSE, DO-WHILE etc.
Structured Programming is something more than this, but it is very difficult to say what it really is, because it does not represent an unequivocally defined entity.
It may be considered to be a number of approaches aiming at the improvement of Software structure.
Thus "Structured Programming" has become a cover-term for everything intended to give a structured form to a Software product. Because of this it is preferable to talk about different methods and techniques and their integration in the programming practice.
It would be however advisable first of all to recall some general foundations.

General Considerations

In general the Software production process can be subdivided into four phases, characterized by the activities of analysis, design, coding and testing.
Since different people may be involved in these four phases, it is important to have available an adequate means of communication in order to avoid misunderstanding of the concepts, on transition from phase to phase.
Furthermore it is important that the structure be outlined as soon as possible, in the whole development process, so the design phase becomes the pre-eminent one.
From the moment of the overall view of the problem, the sketched program must be a functional whole, entirely specified by a set of general functionalities. Each functionality, at this global level, will be subsequently described in detail, which constitutes the next refined level.
Such a process must continue until a set of simple functions, called primitives, is defined. This is the lowest level of refinement.
Thus a hierarchy of levels is established. If each level is expressed by a rigorous description and if one supposes that a family of abstract machines exists, then each level may be assimilated to a version of the program.
Such versions can be verified in the logic, since the control flow is easy to follow. Furthermore one can check whether or not the described actions comply with the original specifications.
In this way the integration of different parts of the program, during the testing phase, occurs with few constraints.

This hierarchical procedure also involves some data organization implications.
Data must also be structured in such a way that all functions defined at one level only have access to the data fields belonging to the same rank within the organization, creating a visibility by level which can be extended to all resources the program requires.
The concept of levels of visibility makes for easy modifiability of any primitive of a certain level, without implications at different levels.
In addition, a logical distinction can occur between global and local variables.
The extended use of local variables increases the protection of the data against unwanted accesses in the case of bugs or failures.
Another general aspect of a structured software is the clarity of the context expressed by the code that becomes more readable.
The readability is mainly obtained by using closed control structures.
However a critical examination of the matter reveals two unfavourable situations that need appropriate remedies.
First of all, there arise many situations which call for multiple exit points from a loop.
The translation of such situations into a form that resorts only to strictly closed structures, requires the introduction of auxiliary variables which combine several conditional events.
To obviate this drawback, additional control structures must be provided, preserving externally a closed control environment, and allowing special exit facilities at different stages of their lay-out.
Secondly, segments of a program that are common to more than one path, cannot be reached from anywhere because of the lack of statements providing the jump possibility. Thus these common segments must be duplicated as many times as the joined paths are.
This situation involves a wastage of core.
A convenient solution can be supplied by a procedural feature providing a linked transfer of control without the computing overhead of the subroutine calls.

Survey of Several Techniques

Top-Down Technique

This is a very incisive technique directly derived from the concept of hierarchical levels. Top-down philosophy can be applied equally well to the design, the implementation and the testing phases.
Starting from the top level, that is a series of general statements which define the global problem, the design proceeds down and supplies a more detailed definition at each lower level.
According to this descending development of the design, the top unit of the program, which accomplishes monitoring functions, can soon be coded and implemented.
This unit can also be tested provided that the functions performed at the lower level are simulated by dummy units, also called stubs.
Such a stub is quickly implemented coding either an empty unit or a unit performing only a control trace by simply writing a message.
No further specifications are needed to do this, because the design of the top level already included all details concerning data exchange to interface with each lower unit (i.e. the calling sequence in the case of subroutine calls).
At the next level, one unit at a time can be implemented and tested replacing, in the existing frame, the identically named stub. This cycle must be repeated until all units of the lowest level are implemented and the whole development process is completed.
Such step by step integration involves only one unit entering the established framework at each new test, and this new unit is first inspected to localize any detected error.
Moreover incorrect units are generally located at their introduction into the evolving environment; detected bugs seldom involve modification of already checked units; higher level units have been widely tested once the total implementation is finished.
The top-down approach can be flexibly adapted to special requirements. In particular, a variation known as "hardest-out" consists in first developing some critical functions on which the problem feasibility may depend, for instance a set of file-handling routines at the lowest level.

Design Media

In general a design medium is a whatever tool, used during the design phase and producing a documented lay-out of the software structure in form of charts or reports.
These documents constitute a means of communication between people cooperating in the software production process, with the resulting advantage of increased understanding of the global process and improved interpretation of the distributed tasks.
It is evident that such means, suitably adapted and edited, can also prove useful in users' training.

Because of this it is implicitly shown that a careful and detailed design brings a methodical contribution to software documentation and it is not surprising that some of the items reviewed are both a design as well as a documentation tool.
Among the design media we find some systematic approaches which meet the standardization exigencies of large working environments.
Some others are more informal and can be adapted to different practices and situations.
Some items of both types are reviewed below.

- The Michael Jackson Design Methodology [7]
  This method is based on the construction of data structures covering all input and output, as the means for a visual representation of the problem; afterwards the program design is modelled on the structured data.
  The method develops in three steps:
  - Step 1 Structure Problem Logic. Starting from requirement specifications, this groups data components and defines relationships between them; visualizes one-to-one correspondences between input and output data structures: in other words, the existence of a direct relationship between a block of needed input data and a block of expected results, in the sense that the later can be derived from the former by a defined transformation.
  - Step 2 Structure Program Logic. This shapes a program component for each shown input/output correspondence. Where such correspondences do not exist, it defines additional blocks of intermediate data to obtain a chain of one-to-one correspondence between data structures, joining input and output. For each generated correspondence a new program component is added.
  - Step 3 Allocate Program Operations. For each program component, this lists the operations to be performed on the data.

  Other general characteristics are: independence from both programming languages and hardware; simplicity of the structures; ease of understanding for every level of programmers.

- The Larry Constantine Structured Design [8]
  The approach followed here is the functional decomposition of the problem in order to give the structure a high modularity. Requirement specifications are represented in graphical form using Data Flow Graphs.
  The information stream flows from left to right crossing some nodes denoting major transformations.
  Each node represents a partial process of incoming arrows to produce outgoing ones and points out a functionality of the problem.

The structure of the program is shown by structure charts which have the form of a descending tree that pictures the links among bounded modules:

- boxes represent the units of the program,
- connections between boxes show the communicating units,
- directed arrows indicate the transfer of shared information.

After this an evaluation of the design is carried out on the basis of two criteria:

- Coupling: this means the degree of connectivity of a module and depends on the number and complexity of connections.
- Cohesion: this means the homogeneity of the actions the module performs. It depends both on the number of tasks and on the relationship between them.

A good design is obtained when both low coupling and strong cohesion are combined.
The design can be partially reworked in order to achieve this goal.
This optimization cycle is one peculiarity of the method.
Moreover the orientation towards functional decomposition favours effective solutions, even if this approach is intuition-based and depends on individual skillfulness rather than on the objective facilities of the method.

- The Pseudo-Codes

The pseudo-code is a simple and manageable tool for rewriting an algorithm in a structured and unambigous natural language.
The structured pattern is given by keywords (usually in capital letters) which are taken from closed control structures reproducing the connective paths between the algorithm primitives, in their turn represented by statements in a natural language (usually in lower-case letters).
When a structure is nested within a more external one, its entire text can be indented so that the internal structure is ranged at the right of the pre-existing alignment.
This identation introduces a sort of two dimensional profile which is an aid in the structure visualization, so that the pseudo-code sheets illustrate the represented algorithm as does the classical flow-chart.
The pseudo-code is widely applicable and can be employed as final means in any design method.

- The Program Design Languages

These are in effect an implementation of a well-defined pseudo-code. First a set of suitable control structures is fixed and a preprocessor checks the text for the completeness of the structures and edits the design in a clean and tidy report.
The relevant aspect of such a Design Language is that it might be the first step in the future trend of Automatic Software Development.
Indeed the output of such a preprocessor might serve as the functional specifications for a Program Generator equipped with powerful capabilities in the field of language analysis and Macro generation.

- The HIPO Design and Documentation Technique [9]

HIPO, which stands for Hierarchy plus Input-Process-Output, is the standardization of a way to illustrate a process that involves data transformations.
The Hierarchy diagram represents as a tree structure the hierarchical levels into which the program design is split.
The Input-Process-Output diagrams illustrate each function of the whole tree by means of three side by side boxes filled by comments.
The central box describes the actions performed by the concerned function. The left-hand box represents the structure of the input data needed by the function as nested squares, while resulting output data are represented in the right-hand box.
Directed arrows connect data with the related action of the central box.
HIPO diagrams aid data-processing documentation and may be of use in exchanging the represented context between different people.
However the extension of HIPO diagrams to the lower levels involves a considerable amount of work updating a lot of diagrams according to the evolving situation.
It must be mentioned that automated production and maintenance of HIPO diagrams can be provided by an IBM package named HIPODRAW. [10]

Closed Control Structures

During the coding phase each statement in natural language produced by the design will be coded in one or more statements of a programming language.
The chosen programming language must provide a number of closed control structures to code the flow of control in the way supplied by the design.
These structures are not standardized and the features allowed change from language to language thus the specific syntax must be observed.
Some general considerations on the most used linguistic constructs are reported later.
It is clear that the forms presented do not refer to any particular language.

- Selection Structures
  The selection of the control between two alternative ways is commonly allowed by the IF construct which can assume different forms according to whether the ELSE clause is absent or present or optional.
  A form which synthesizes all these situations is:

```
IF <condition> THEN <block1> [ELSE <block2> ] ENDIF
```

  A variation is possible, including any number of clauses ELSEIF before the clause ELSE. This form results in a single structure instead of a series of nested IF structures.

```
IF<condition>THEN<block1>ELSEIF<block2>... [ELSE<block n>]
ENDIF
```

  For instance the two codes of Example 1 implement the same algorithm.

  EXAMPLE 1

```
Code A                  Code B

IF p                    IF p
THEN                    THEN
   a                       a
ELSE                    ELSEIF q
   IF q                    b
   THEN                 ELSEIF r
      b                    c
   ELSE                 ELSE
      IF r                 d
      THEN              ENDIF
         c
      ELSE
         d
      ENDIF
   ENDIF
ENDIF
```

The selection of one among several ways is allowed by a CASE construct.
The simplest form of this construct is commonly represented by:

```
CASE<arithmetic expression> <block 1>   <block 2>...  <block n>
ENDCASE
```

Such a construct operates as follows. The <arithmetic expression> is evaluated; supposing that the integer part of the result is i , if $1 \leq i \leq n$ then the <block i> of statements will be executed otherwise none of the listed blocks of statements will be executed.
A more genefalized form allows the selection of a block to be executed when the result of any expression lies between a case list.
The keyword used for this form is either CASE or SELECT:

```
CASE <expression> <case list 1> <block 1>
                  <case list 2> <block 2>
                  ...
                  <case  list  n> <block n>  [ELSE <block n+1>]
ENDCASE
```

where <case list i> is a list of possible values that <expression> can assume, causing <block i> to be executed.
Conversion is allowed between compatible types of the result of <expression> and the case value.
If the result does not match with any case within all case lists, the block included in an optional ELSE clause will be executed.
A further development of this structure is represented by an "event-driven" CASE that is a very powerful construct being a combination of iteration and selection of control. For this reason it will be described after the iterative constructs.

- Iteration Structures

  The basic iteration structure is the WHILE-DO which may be presented in the form:

  ```
  WHILE <condition> DO <block> ENDWHILE
  ```

  In some implementations the keyword DO is omitted.
  This structure implies the following operations: Firstly the <condition> is tested, if proved false the entire construct is skipped otherwise the <block> of statements will be executed and the whole process repeated.
  This cyclic operation is finished and the construct left at the first failure of the <condition>.
  That is to say what is clearly expressed by the keywords of the construct; the <block> of statements is repeatedly executed while the <condition> remains true.
  In some cases the parameters of the condition have to be defined by a previous execution of the block of statements:

  ```
  <block> WHILE <condition> DO <block> ENDWHILE
  ```

This situation is better represented by a variant of the construct known as DO-UNTIL:

```
DO <block> UNTIL <condition>
```

where the postposition of the keyword UNTIL permits the omission of an ending keyword. It should also be observed that the keyword DO is replaced by REPEAT in many implementations.
This construct works in the following way; the <block> of statements is executed and afterward the <condition> is tested, if proved true the construct is left otherwise it is entered again with a new execution of the <block> and so on. The cycle is broken off once the <condition> is achieved. Briefly the <block> of statements is executed repeatedly until the <condition> becomes true.

The repetition of some operations on array elements, implying the increase of a subscript variable is facilitated by the following structure:

```
FOR <variable> <value1> <value2> [<value3>] DO <block> ENDFOR
```

where <value1> is the initial value assigned to <variable>, <value2> is the upper limit that <variable> may assume and <value3> is the increment to be added to <variable> at each new cycle. When <variable> must be increased by one, <value3> may be omitted. Some implementations omit the keyword DO.
But none of these constructs allows the condition test between two blocks of statements to be executed repeatedly, without duplication of code.
A generalized LOOP structure has been proposed as being the best solution for this: [11]

```
LOOP <block1> WHILE <condition> <block2> REPEAT
```

Moreover this structure simulates the WHILE construct when <block1> is omitted and the UNTIL construct when <block2> is in its turn omitted.
Example 2 shows a comparison of the three structures in the case of exit between two blocks, while Example 3 shows the use of the LOOP structure in the classical "WHILE" and "UNTIL" situations.

EXAMPLE 2

Code A

```
LOOP
  a
WHILE not(p)
  b
REPEAT
```

Code B

```
a
WHILE not(p) DO
  b
  a
ENDWHILE
```

Code C

```
DO
  a
  IF not (p)
  THEN
    b
  ENDIF
UNTIL p
```

Since the predicate p represents the exit condition, its complement is used to mean the 'stay-in-loop' condition.

EXAMPLE 3

Code A

```
LOOP WHILE q
   a
REPEAT

LOOP
   a
WHILE not(r) REPEAT
```

Code B

```
WHILE q DO
   a
ENDWHILE

DO
  a
UNTIL r
```

- Jump Capabilities

Since the blocks of statements may be as complex as possible, including multiple nested control structures, conditional events through this path may make it necessary to leave the construct before the end of a block is reached. Such a situation is marked by the number n of potential exit conditions.
An effective representation of this generic Ωn path implies the need of some disciplined jump capability.
While the use of closed structures for the iteration of the control has introduced a discipline for jumps backwards, this new required capability must govern jumps forwards.
A primary solution was the introduction of an EXIT clause within the iteration construct. Such a clause, wherever inserted, allows the transfer of the control to the first statement after the ending keyword of the inner iterative structure that contains it.
A new problem arises when nesting two or more structures of this type. Indeed the simple EXIT makes it possible to leave one iterative path, that is a level of the control flow. But some events (e.g. the detection of severe errors) may imply leaving an outer iterative structure, in other words to jump through more than one level.

To this purpose a multi-level jump was included in some proposals by means of an EXIT <n> clause.
Such an extension of the demonstrated need for a jump is not exempt from negative implications. Indeed multilevel EXIT is extremely dangerous because it establishes a static jump without any contextual relationship.
This fact precludes any compile-time verification, therefore modifications and extensions to the software may become difficult, if not impossible, as happened before.
For instance, in the case of data manipulation by levels, each of them involves a clear definition of data fields, primitive functions and operators, all detailed in homogeneous parts of a structured code. Data consistency is checked and any detected failure causes the interruption of the current operation. If, at any moment a new intermediate level has to be added, all the pre-existing EXIT n clauses in the lower levels must be revised to avoid troubles at execution time.
Analogous trouble is caused by the suppression of an existing level.
Another solution is represented by the introduction of a LEAVE clause that allows some labelled closed structure to be left transferring the control to the first statement after the ending keyword of the referred entity.
This clause may be valid only in a portion of code which represents a functional unit (procedure, block or similar units) and has the form

```
          LEAVE   <label>
```

whose semantics is related to the occurrence and the location of <label> within the unit.
Thus the causes of errors such as duplicated, misplaced or missing labels can be checked at compilation time. To explain the different behaviour of the two clauses, let us imagine a block of statements composed of three nested closed structures labelled 'alpha', 'beta', 'gamma' from outside to inside and representing as many hierarchical levels. The inmost structure 'gamma' performs a validity check and when an error occurs the control must leave the whole block.
The desired jump may be obtained by a clause

```
          LEAVE   alpha
```

whose effect is equivalent to EXIT 3 in a fixed situation and will not be affected by any addition or suppression of intermediate levels.
Since the LEAVE clause involves a restricted use of labels, namely a closed structure label, it appears rather natural that the needed jump capability may also be provided by a restricted use of a GO TO clause within the iteration constructs.
Indeed the relevant thing is the ability to make a jump, and not the linguistic means to provide it. However, it is necessary to emphasize the exceptional nature of the jumps, and to recommend restricting them to the control of abnormal paths.

Jumps must never exceed the scope of a procedure, therefore when levels are implemented as procedures, since such units must not make assumptions upon the external environment, the most reliable solution is the use of a 'case variable' as parameter. This variable records the casual occurrence of one among some possible events, and returns this value which will be tested outside the procedure in a CASE structure.

- Event-Driven Structure

  The concept of this recent structure [12] may be represented in the form:

```
UNTIL  <event 1> OR <event 2> OR...OR <event n>
    <block 0>
REPEAT
THEN   <event 1> <block1>
       <event 2> <block 2>
        ...
       <event n> <block n>
END
```

  The <block 0> will be repeatedly executed until the occurrence of one of the foreseen n events, then the occurred <event i> determines the selection of the <block i> to be executed and finally the control leaves the structure. It must be noticed that one of the n events must assume a special meaning, namely the avoidance of the loop for ever. This safety event may be tied to the execution of a finite number of cycles or, in the case of parallel execution of concurrent processes, it may be an external signal.

Software Production Libraries

A software system being implemented by the top-down method evolves continuously.
The task of fixing a set of homogeneous versions is made possible by using some production libraries to store both source code and test data. The essential requirement is to have:

- a Master Library which contains the overall up-to-date version of the system
- a Development Library containing some newly implemented system components or modified ones
- a Test Library containing the case data used to test the system

Each fully tested component will replace the identically named stub in the Master Library in the same way that modified components replace their previous version. The data used for testing will be added to the Test Library.

It is advisable, at each modification, to run again all case data stored in the Test Library.
Furthermore some old versions of the system at well tested stages may be stored as a family of static libraries.
The management of these production libraries is performed automatically by various information filing systems or librarians.

Managerial Framework

In the case of large scale projects, the improvement of programming productivity also depends on the standardization of the methods and tools employed as well as a rational organization of the programmers' activities.

- Chief Programmer Team
  This is the most well-known management approach in the programming environment.
  It consists of task separation, high specialization and disciplined relationships amongst team members.
  It is important to individualize the following taks:
  - Direction: the responsibility of design and implementation of the most critical components as well as the supervision of the development and the integration of the system;
  - Assistance: the need to cooperate in the development of critical components;
  - Maintenance: the updating of the support libraries and documentation, together with the annotation of the project progress in journals;
  - Operation: development, coding and testing of single components.

  These specialized tasks may be assigned to different members of a team or grouped to be carried out by a few people or one individual according to the manpower available.
  In the pilot stage [13] the programmer team was managed by a nucleus of three members:
  - the chief programmer, who assures the direction and is the primary decision-maker;
  - the back-up programmer, responsible for assistance, having also a share in all important decisions, and who may replace the chief programmer if need be;
  - the programming secretary, who attends to maintenance and coordinates all routine work such as run requests and listing registration.

  The other team members, up to three programmers, are responsible for operation and report directly to the chief programmer.

  The first achievements announced were very promising and a generalization of this approach would be suitable for the promotion of a Software Factory.

- Structured Walk-throughs
  This is a management control based on periodic reviews during the design phase. At each review a well defined component of a project is focused.
  The person responsible for the development of this component reports on specifications, assumptions and interfaces with other components of the project. Everyone involved in the system development as well as managers concerned with the project may attend the meetings and discuss the details of the review.
  These critical discussions increase the project awareness among participants and will promote a better impact of the software at production time.
  This practice may also be continued during the coding phase, but in this case the participation is restricted mainly to the programmers.

A Style as Integration of Different Techniques

Although developed almost independently, the techniques reviewed share the common goal of contributing to the improvement of Software structure.
Some of them partially overlap as far as either the explored domain or the adopted solution are concerned.
In this perspective some perplexity may exist in the choice of the method best suited to any single case.

A balanced solution consists in the integration of some selected techniques, according to the extent of the planned Software product.
An attempt at integration is proposed for application by the individual programmer.
Of course, there is no point in making a list of strict rules to be followed blindly.
Programming must be an imaginative activity, therefore programmers must carry out their creative task in a manner based on functional principles and tailored to their own capabilities. Personal intuition must be stimulated to invent original alternatives in finding the solution to a given problem.

The topics presented from now on are intended as suggestions to guide people, firstly in formulating a solution and secondly in choosing between the different possible approaches to a solution, so that the best one will be adopted.
This is the way to establish a sound programming style.

The proposal is explained by a simple example. Though the problem is trivial, the operating method can be extended up to medium size Software products.

"What is medium size?" is the obvious question at this point.

A subjective classification of Software products depending on their size might be:

- the small ones concern trivial or didactic problems;
- the medium ones concern complex problems developed within a reasonable time by one programmer;
- the large ones concern problems worked out by a team because of their extent or urgency.

Let us suppose that we are faced with a problem involving some generic computations on an unpredictable number of cases, each of them represented by a set of input data.
The last input information of each set being an indicator which tells whether another set follows or not.
In addition to the specific input data the computations require some reference information contained in a data library.
Either the results or an appropriate error report will be printed for each case.

In the early stages it is necessary to consider the data organization.
Since any informatics process may be seen as a transformation upon some information, the delimitation of self-contained functions may be deduced from logical connections within the information.
Therefore the identification of such connections as well as the recognition of relationships between input data and output results, allows a rational data organization, on which the frame of the design will be based. (See the concept of the Jackson method [7]).

Our example deals with:

- Reference data stored in an established library.
  First of all the extent of the library governs the core requirement as well as the choice between a single access to the whole library or periodical retrieval of one needed subset of it.
  Secondly, its content affects both the attributes of the variables which are going to receive the data, and the mode of reading them in.
- Specific input/output data, whose composition and complexity will be a guide to the choice of proper variable structures to store them.
- Error types, on which the message texts will depend.

We assume that:

- the library is short enough to be kept entirely in core,
- each set of input data contains: a descriptive title, a hierarchy of numeric fields, a logical indicator,
- the results ensuing from each input set are requested in form of a table,
- errors are grouped in two classes that are:
    - inconsistent input
    - computation failures

A preliminary analysis of the requirements points out:

- the nature of the processing (computation on an unpredictable number of cases)
- the information required (specific data contained in each input set in conjunction with reference data always valid)
- the expected information (printout of results or error report obtained by the processing of each input set)

The broad lines of the design emerge from these general elements. Some actions stand out as well as their relational occurence.

Action	Occurrence
get reference data	once, before all other actions
get specific input data	for each set
check data consistency	for each set, after data are got
make computations	only for valid sets
check for completion of computations	after computations on a valid set
print results	only for correctly processed sets
prepare and print message	when any error has occurred
stop the processing	at the end of the last set

The reported actions must be linked according to the logic emerging from the relationships between their occurrences.
Such linked actions constitute a picture of the top level of the problem solving process.
The suggested tool to describe this abstract process is a pseudo-code.
A relevant consideration must be made before defining or choosing the pseudo-code to be used. It is generally admitted that a distinctive feature of a good design is to be language independent. However, it is an undeniable advantage if the same control structure layout is present in both the pseudo-code and the programming language.
Indeed, in this case, the coding activity will mainly consist in replacing action descriptions with statements of the programming language, while the control flow is already settled, the pseudo-code keywords being valid also in the programming language. So this derogation to the principle of language independence seems to be acceptable.

```
PROGRAM
   Get reference data from Library
   DO
     Get one input set and check data consistency
     IF consistent data
     THEN make computations
          IF computations completed
          THEN print results
          ENDIF
     ENDIF
     IF any error
     THEN prepare and print messages
     ENDIF
   UNTIL no input set follows
END PROGRAM
```

The program in pseudo-code shows some statements in natural language that describe the logical functions into which the program itself is decomposed.

In the continuation of the text, such functions will be referred to by more concise identifications. The table below gives the correspondences.

Get library	- Get reference data from library
Get and check	- Get one input set and check data consistency
Compute	- Make computations
Results	- Print results
Messages	- Prepare and print messages

At this point we deal with the problem of data visibility, that is the need to control the access to data that must be shared by different functions.
The following summary shows the situation in our problem:

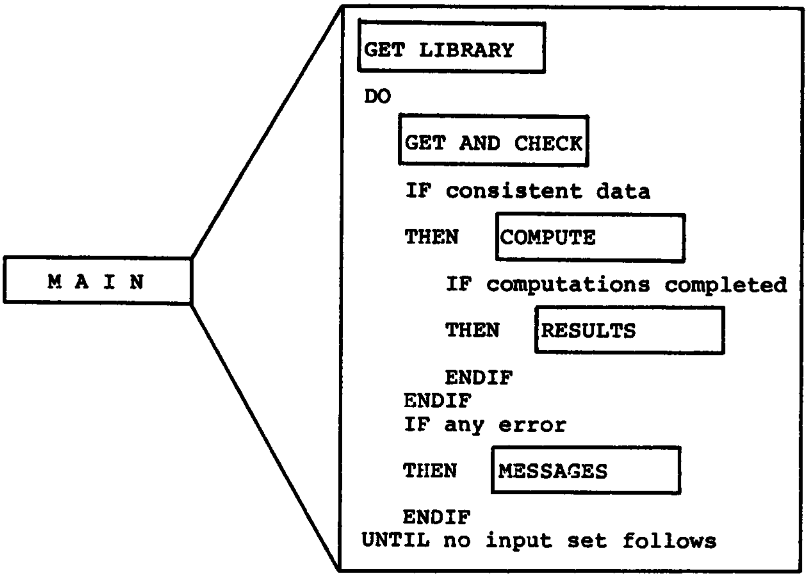
Accessed data	Functions
reference data	from: - Get library, - Compute
specific input data	from: - Get and check, - Compute
output results	from: - Compute, - Results
error indicators	from: - Get and check, - Compute, - the top unit
stop indicator	from: - Get and check, - the top unit

In this way the interfaces between communicating functions are clearly defined.

Now the design will proceed from the top level which is completely expressed, down to the more detailed levels, operating in the same way on each separate function.
At this point one cannot go into details, because of the generic formulation of the problem. We can only remember that existing data structures are a tangible element on which the architecture of the function itself may be based. For instance the function 'Get and check' will be tailored to the hierarchy of input data.
When the design is completed the problem must be coded in a programming language.
The work to be done in this phase greatly depends on the features of the chosen language, so that only some suggestions will be given here.
Some implementation decisions must be taken before starting to code the program.
Provided that the language allows independent compilations, a modular scheme of the program must be planned. This means that generalized and close functions are identified as independent units to be compiled separately. (Procedures or subprograms according to the terminology of the language)

Although it is not a general rule to consider each function as an independent unit, we do choose this way to emphasize the concept of program segmentation as well as the need to keep the top unit simple and clear.

Thus all existing functions become independent units and we just add to them a top unit called MAIN that simply monitors the other ones, and the program scheme stands out as follows:



```
M A I N

GET LIBRARY
DO
    GET AND CHECK
    IF consistent data
    THEN COMPUTE
        IF computations completed
        THEN RESULTS
        ENDIF
    ENDIF
    IF any error
    THEN MESSAGES
    ENDIF
UNTIL no input set follows
```

Now, for each unit, one must provide:
- declaration of all local variables,
- declaration of the variables shared with other units using the facilities allowed by the language to set up interfaces between units,
- executable statements performing the required actions.

Applying the top-down technique the testing may start as soon as the MAIN unit is implemented and proceeds to each new unit implementation. Thus the problem will be tested step by step.

At the first step only the MAIN unit exists, while all other functions are temporarily simulated by stubs that are dummy units whose only goal is to show a trace of the control. Therefore the first action of a stub is to write a message to prove that the unit has been entered; moreover the stub will print the value of any parameters transferred to the simulated unit by the calling one.
Thus the first run only produces a figure of the chronological link of the different units.
The next steps must provide the test of one unit at a time.

The functions to be tested first are, respectively, reading input data and editing the results. Indeed transformations upon data cannot be verified without the certainty that data are correctly accessed and that the results are printed exactly.
The first verification on data is to print the values that have just been read.

Obviously the unit GET LIBRARY will always read the same data, therefore the operations to write the accessed values may be removed as soon as the test is successful. However it is advisable to maintain such operations also in the final version of the unit GET AND CHECK, because this may also be useful for the debugging of anomalous behaviour of the program in the exploitation phase.
The test of the unit RESULTS implies the availability of a set of suitable values stored in some arrays. This may be obtained, for instance, by a series of assignment statements in the COMPUTE stub. In this way, also the access to the arrays shared by the two units will be tested.
The units GET AND CHECK and COMPUTE that involve alternative situations must be tested both for the valid alternative, and for all the foreseen errors.
The test of the unit MESSAGES must just be focused on the data which represent these abnormal situations to be sure that the proper message is issued for any foreseen error.
The simulation of a complete set of error conditions is very important for obtaining reliable diagnostics during the real work of the program.
The evolution of the program during the testing phase is represented by the following steps:

Step 1

coded unit	to test:
MAIN	- loop over multiple input sets
stubs	to simulate:
GET LIBRARY	- library transfer
GET AND CHECK	- input transfer
COMPUTE	- computations
RESULTS	- editing of results
MESSAGES	- issue of error messages

Step 2

coded units	to test already tested functions plus:
MAIN	
GET AND CHECK	- transfer and validation of input data
stubs	to simulate:
GET LIBRARY	- library transfer
COMPUTE	- computations
RESULTS	- editing of results
MESSAGES	- issue of error messages

Step 3

coded units	to test already tested functions plus:
MAIN	
GET LIBRARY	- library transfer
GET AND CHECK	
stubs	to simulate:
COMPUTE	- computations
RESULTS	- editing of results
MESSAGES	- issue of error messages

Step 4

coded units	to test already tested functions plus:
MAIN	
GET LIBRARY	
GET AND CHECK	
RESULTS	- editing of results
stubs	to simulate:
COMPUTE	- computations
MESSAGES	- issue of error messages

Step 5

coded units	to test already tested functions plus:
MAIN	
GET LIBRARY	
GET AND CHECK	
COMPUTE	- computations
RESULTS	
stub	to simulate:
MESSAGES	- issue of error messages

Step 6

coded units	to test already tested functions plus:
MAIN	
GET LIBRARY	
GET AND CHECK	- rejection of input inconsistencies
COMPUTE	- failures in computations
RESULTS	
MESSAGES	- issue of error messages

After the last step all the units are integrated and the whole program is tested.

The peculiarity of such a descending path is that the monitoring of the global logic is the most tested part of the program.

In conclusion, some key suggestions may be summarized.
During the design phase:
- always have in mind the global view of the problem,
- outline the organization and visibility of all data before thinking about their transformation,
- divide complex functions up into simpler primitives,
- describe the actions to be performed, using concise and unequivocal statements in natural language.

During the coding phase:
- find the best segmentation into independent units,
- declare all variables and restrict the use of global variables to essentials.

During the testing phase:
- for each level of abstraction, implement crucial units beforehand, so that they can be tested more thoroughly,
- store all test cases and each time a unit has been modified, rework all cases pertaining to it.

A TALK ABOUT PROGRAMMING LANGUAGES

The development of the languages for computer programming has passed from the machine level to stages that more closely match the user's need.
So we have had several classes of languages called from time to time, high level languages, problem-oriented languages, very high level languages etc., and the number of languages is continuously growing, especially in the academic environment.
Because of the size of the subject, a general discussion on programming languages exceeds the scope of this paper, and consequently this chapter will only concern general-purpose languages widely used for scientific applications.

The Ideal Programming Language

It frequently happens that programmers are restricted by some lack of the language used.
By collecting such negative observations, it is possible to infer a set of the desirable features that a suitable language should have.
First of all the syntax of a programming language should be formally defined.
This preliminary qualification allows a coherent implementation of compilers by different computer manufacturers. Moreover a formal definition represents an unequivocal basis for systematic approaches in teaching computer programming.
A lot of important features follow.
Their natures differ widely, so it is more convenient to group them according to some general subjects.

a) Syntactical and editing facilities:
- extended character set;
- free format for the source statements;
- long identifiers for the symbolic elements of the language, with a connecting character like hyphen, dot or underscore. By this, meaningful names are easily allowed giving more understandable codes;
- both in line and interspersed comments. The former are for concise explanations of single operations, the latter for descriptive documentation of blocks of code;
- automatic indentation which illustrates the logic by showing up nested levels.

The last three features are very useful in obtaining a self-documented source code.

b) Data representation:
- parameter variables, i.e. symbolic names for constants, may improve readability;
- flexible facilities to handle character strings of variable length;
- extendable data type definition to support data abstraction creating convenient new types;
- multidimensional arrays for effective representation of compact matrices of homogeneous data;
- composite structure for flexible representation of chains of heterogeneous data, as well as sparse matrices;
- dynamic allocation may be a useful option, indeed it is a powerful feature, but its thoughtless use should be avoided because it involves time-consuming algorithms for core management.

c) Management of control:
- a complete set of closed structures, avoiding both uncontrolled branching structures and multiple entries;
- a macro generation mechanism for in-line expansion of a short portion of code needed in several places. This feature is especially suitable for time optimization because it involves only a sequential execution of the embedded instructions;
- internal procedures to be referred to from everywhere within a program unit by a simple link mechanism and without space duplication. Since the linkage may be implemented without any noticeable overhead, this feature may be an alternative to the previous one, but more oriented to space optimization;
- external procedures allowing independent compilation. This feature permits a physical decomposition of the program into distinct modules and, in spite of the execution overhead involved, is of wide application to support external libraries;
- recursive procedures as an optional feature is of great interest. Indeed recursion is a very clear and synthetic means of expressing algorithms. But users must be very careful not to fall into abuse of it because the process cannot be implemented efficiently. On this subject, it must be remembered that advanced techniques for program optimization comprise transformation from recursion to iteration [11].

d) Debugging facilities:
- a complete set of intelligible diagnostics of the syntax errors detected by the compiler;
- a powerful set of error handling procedures to issue explanatory messages and explicit report of recovering actions if any, whenever the running code is discovered to be behaving abnormally;
- a wide set of tracing facilities to prompt significant information as soon as a definite bug occurs.

e) Cost-effective facilities:
   - a data management system supporting a wide range of peripheral devices and able to cover any access mode, aiming at both efficiency and device independence;
   - specialized public libraries that must be efficient, reliable and well protected;
   - private libraries to be concatenated to the public ones and easily extendable by user-defined procedures or macros;
   - official definition of a standardized subset to guarantee reliable performances on different compilers;
   - standard conventions for interface definition as well as the recognition of linkage conventions of other languages, so that connections between external modules coded in different languages become possible.

It should be observed that points a) to c) include the most profitable features in order to produce a readable, modifiable and maintainable code which is expected to be as little error prone as possible. However the last two points must not be neglected because they respectively make provability easier and allow efficiency. Experience has also shown that the success of a programming language greatly depends upon these features.

Using the Existing Languages

Unfortunately the best known languages, and maybe any general-purpose language, do not combine all the features we need to apply the described techniques. So we must adapt our habits to the adopted tool.

Algol-like languages, for instance SIMULA 67, ALGOL 68, PASCAL etc., provide extended means for data representation but in some cases the implicit mechanism of global variables may hide unexpected side-effects.
Control structures are generally well suited to build closed control environments and nest them properly, but uncontrolled branches are to be avoided.
The program segmentation is mainly achieved by internal procedures, including recursion. A systematic verification of data type compatibility concerning actual and formal parameters is performed. Furthermore scalar parameters may be transferred by value, increasing data protection.
However the use of external procedures is made possible in some cases. For instance the CDC 6000 PASCAL compiler includes two declarations to refer to external libraries [14]: EXTERN for PASCAL procedures and FORTRAN for library subprograms compiled by a FORTRAN processor. This later possibility might make of PASCAL the modern programming language which can enrich its range with the patrimony of decades of programming efforts.

Due to its supple and powerful data type definition mechanism, PASCAL may be considered an extendable language and its growing significance is confirmed by the fact that the U.S.A. Department of Defence is supporting a study to define a new common scientific oriented programming language based on PASCAL [15].
Two negative points must be noted however, for this family of languages: Input-output facilities are rather poor and debugging is not always well supported.

The PL/I checkout and optimizing compilers provide good data representation, clear control structures and the full set for program segmentation: macros plus internal, external and recursive procedures.
In spite of this excellent equipment trouble is very frequently met.
The user must be aware that side-effects against data security, may occur by use of global variables and because of the transfer of parameters which takes place by address.
In addition unclosed control structures as well as multiple entries should be avoided.
Input-output offers extended possibilities, while the sophisticated recovery facilities are of help only to highly experienced users.

A special word will have to be given on FORTRAN IV.
This language is lacking in several features which are indispensable to the building of well-structured programs. Indeed, data structure are limited to arrays and closed control structures are inexistent. Moreover scalar parameters are transferred in a hybrid way which does not assure data protection, i.e. they are passed by value, returning result when the subprogram terminates.
Nevertheless FORTRAN cannot be ignored, because of its importance from an economic point of view, ensuing from execution time optimization, independent compilation, simplified but efficient input-output and effective debugging.
The use of a precompiler has been proposed as a remedy.
But the proliferation of such tools has pointed out the weakness of the enterprise.
The overall precompiler tangle can be summarized by the sentence "It is a good thing they exist, but it is a pity we needed them to exist". In fact these tools have made it possible, and still do, for people in envirorments lacking in proper languages, to be intiated to and trained in structured coding. Some of them first implemented automatic indentation, still missing in programming languages. Above all their existence engendered in the FORTRAN world the need for closed structures and consequently promoted analytical studies to define coherent and standardized proposals [16] which should hopefully result in an extension in this direction of the Language and the compilers as well.

But their generalized use is causing us to lose sight of the primary benefits of a compiler: effectivness, direct path in debugging, compatibility with the operating system and maintenance supported by the manufacturers, etc.

In conclusion the best approach, using FORTRAN IV, seems to be a great effort to emphasize a clear and well-structured design, followed by a disciplined use of fixed control construct which simulate closed structures by means of the control statements of the language.
Proposals along these lines can be found in the literature [17], [18], but any programmer can devise clever solutions, read this text with criticism and hopefully find something to his profit.

Acknowledgements.

I wish to thank Mr. Jean Pire whose suggestions helped me to improve the text and make it clearer.
Furthermore I am greatly indebted to Mr. Giancarlo Gaggero, who gave me the opportunity, late in 1974, to learn and afterwards to apply the structured techniques, and to all cited authors whose papers were a continuous guide in my work.

REFERENCES

[1] KNUTH, Donald E. "Computer Programming as an Art"
Comm.ACM 17,12(Dec 1974) pp 667-673

[2] BOOLE, George "An Investigation of the Laws of Thought, on which are Founded the Mathematical Theories of Logic and Probabilities" (1854)

[3] HOARE, C. Anthony R. "An Axiomatic Basis for Computer Programming"
Comm.ACM 12,10(Oct 1969) pp 576-580,583

[4] KLEENE, Stephen C. "Representation of events in nerve nets and finite automata"
Automata Studies (Annals of Mathematics Studies, n. 34) Princeton (1956)

[5] DIJKSTRA, Edsger W. "Go To Statement Considered Harmful"
Comm.ACM 11,3(Mar 1968) pp 147-148

[6] BÖHEM, Corrado; JACOPINI, Giuseppe "Flow Diagrams, Turing Machines And Languages With Only Two Formation Rules"
Comm.ACM 9,5(May 1966) pp 366-371

[7] Infotech International Limited "The Jackson Design Methodology" - Publication of the series Infotech Structured Design Division

[8] STEVENS, Wayne P.; MYERS, Glenford J.; CONSTANTINE, Larry L.
"Structured Design" IBM System Journal 13,2(1974) pp 115-139

[9] STAY, James F. "HIPO and Integrated Program Design"
IBM Systems Journal 15,2(1976) pp 143-154

[10] International Business Machines Corporation
"HIPODRAW A Productivity Aid for automated production of HIPO diagrams"
IBM order number G320-5546

[11] KNUTH, Donald E. "Structured Programming with Go To Statements" Computing Surveys 6,4(Dec 1974) pp 261-301

[12] ZAHN, Charles T. "A Control Statement for Natural Top-Down Structured Programming" presented at Symposium on Programming Languages, Paris, April 9-11,1974. Published in Lecture Notes in Computer Science n.19 Springer-Verlag (1974) pp 170-180

[13] BAKER, F. Terry; MILLS, Harlan D. "Chief Programmer Teams" Datamation 19,12(Dec 1973) pp 58-61

[14] MOHILNER, Patricia R. "Using Pascal in a Fortran Environment" Software-Practice and Experience 7,3 (Jun/Jul 1977) pp 357-362

[15] Department of Defense U.S.A. "Requirements for High Order Computer Programming Languages" Revised IRONMAN (Jul 1977)

[16] MEISSNER, Loren P. "Proposed Control Structures for Extended Fortran" SIGPLAN Notices 11,1(Jan 1976) pp 16-21

[17] TENNY, Ted C. "Structured Programming in Fortran" Datamation 20,7(Jul 1974) pp 110-115

[18] International Business Machines Corporation "An introduction to Structured Programming in Fortran" IBM Order Number GC20-1790-0

BIBLIOGRAPHY ORIENTED TO THE ON SITE LIBRARIES

A) Programming Formalization

- CLINT, M.; HOARE, C.A.R. "Program Proving: Jumps and Functions"
  Acta Informatica 1,3(1972) pp 214-224

- ELGOT, C.C. "Structured Programming With and Without GO TO Statements"
  IEEE Transactions on Software Engineering SE-2,1 (Mar 1976) pp 41-53

- HANTLER, S.L.; KING, J.C. "An Introduction to Proving the Correctness of Programs"
  Computing Surveys 8,3(Sep 1976) pp 331-353

- HOARE, C.A.R. "Proof of a Program: FIND"
  Comm.ACM 14,1(Jan 1971) pp 39-45

- KATZ,S.; MANNA,Z. "Logical Analysis of Programs"
  Comm.ACM 19,4(Apr 1976) pp 188-206

- LANZARONE, G.A.; ORNAGHI, M. "Program Construction by Refinements Preserving Correctness"
  Computer Journal 18,1(Feb 1975) pp 55-62

- MANNA, Z.; PNUELI, A. "Axiomatic Approach to Total Correctness"
  Acta Informatica 3,3(1974) pp 243-263

- MANNA, Z.; SHAMIR, A. "The Optimal Approach to Recursive Programs"
  Comm.ACM 20,11(Nov 1977) pp 824-831

- SOKOLOWSKI, S. "Axioms for Total Correcness"
  Acta Informatica 9,1(1977) pp 61-71

See also references [3] and [6]

B) Program Design

- COHEN, A. "New Trends in Program Design"
  Computer World 6,9(1975) pp 2-6

- FROST, D. "Psychology and Program Design"
  Datamation 21,5(May 1975) pp 137-138

- GEORGE, R. "Eliminate Flowchart Drawings"
  Software-Practice and Experience 7,6(Nov/Dec 1977)
  pp 727-732

- HOLTON, J.B.; BRYAN, B. "Structured Top-Down Flowcharting"
  Datamation 21,5(May 1975) pp 80-84

- JACKSON, M. "Structured Program Design"
  Software World 7,3(1976) pp 2-6

- McCLURE, C.L. "Top-Down, Bottom-Up, and Structured
  Programming"
  IEEE Transactions on Software Engineering SE-1,4
  (Dec 1975) pp 397-403

- MILLS, H.D. "Software Development"
  IEEE Transactions on Software Engineering SE-2,4
  (Dec 1976) pp 265-273

- MYERS, G.J. "Composite design facilities of six programming
  Languages"
  IBM System Journal 15,3(1976) pp 212-224

- MYERS, G.J. "Characteristics of composite design"
  Datamation 19,9(Sep 1973) pp 100-102

- PARNAS, D.L. "A Technique for Software Module Specification
  with Examples"
  Comm.ACM 15,5(May 1972) pp 330-336

- PARNAS, D.L. "On the Criteria To Be Used in Decomposing
  Systems into Modules"
  Comm.ACM 15,12(Dec 1972) pp 1053-1058

- PETERS, L.J.; TRIPP, L.L. "Is Software Design "Wicked"?"
  Datamation 22,5(May 1976) pp 127-136

- VAN LEER, P. "Top-down development using a program
  design Language"
  IBM Systems Journal 15,2(1976) pp 155-170

- WIRTH, N. "Program Development by Stepwise Refinement"
  Comm.ACM 14,4(Apr 1971) pp 221-227

- WITTY, R.W. "Dimensional Flowcharting"
  Software-Practice and Experience 7,5(Sep/Oct 1977)
  pp 553-584

See also references [8] and [9]

C) Analysis of Control Environments

- ADAMS, J.M. "A General, Verifiable Iterative Control Structure" IEEE Transactions on Software Engineering SE-3,2 (Mar 1977) pp 144-149

- BOCHMANN, G.V. "Multiple Exits from a Loop without the GOTO" Comm.ACM 16,7(Jul 1973)

- DONALDSON, J.R. "Structured Programming" Datamation 19,12(Dec 1973) pp 52-54

- GOODENOUGH, J.B. "Exception Handling: Issues and a Proposed Notation" Comm.ACM 18,12(Dec 1975) pp 683-696

- LEDGARD, H.F.; MARCOTTY, M. "A Genealogy of Control Structures" Comm.ACM 18,11(Nov 1975) pp 629-639

- LIPTON, R.J.; EISENSTAT, S.C.; DEMILLO, R.A. "Space and Time Hierarchies for Classes of Control Structures" Journal ACM 23,4 (Oct 1976) pp 720-732

- MILLER, E.F.jr; LINDAMOOD, G.E. "Structured Programming: Top-down Approach" Datamation 19,12(Dec 1973) pp 55-57

- PETERSON, W.W.; KASAMI, T.; TOKURA, N. "On the Capabilities of While, Repeat, and Exit Statements" Comm.ACM 16,8(Aug 1973) pp 503-512

- WEGNER, E. "Tree-Structured Programs" Comm.ACM 16,11(Nov 1973) pp 704-705

- WIRTH, N. "On the Composition of Well-Structured Programs" Computing Surveys 6,4(Dec 1974) pp 247-259

See also references [5] and [11]

D) Application of Closed Environments

- ELSHOFF, J.L. "The Influence of Structured Programming on
PL/I Program Profiles"
IEEE Transactions on Software Engineering SE-3,5
(Sep 1977) pp 364-368

- International Business Machines Corporation
"An Introduction to Structured Programming in PL/I"
IBM Order Number GC20-1777-1

- LEDGRAD, H.F.; CAVE, W.C. "Cobol Under Control"
Comm.ACM 19,11(Nov 1976) pp 601-606

- MIZE, J.L. "Structured Programming in COBOL"
Datamation 22,6(Jun 1976) pp 103-105

- NOONAN, R.E. "Structured Programming and Formal Specification"
IEEE Transactions on Software Engineering SE-1,4
(Dec 1975) pp 421-425

- RALSTON, A.; WAGENER, J.L. "Structured Fortran - An Evolution
of Standard Fortran"
IEEE Transactions on Software Engineering SE-2,3
(Sep 1976) pp 154-176

- RIEKS, G.E. "Structured Programming in Assembler Language"
Datamation 22,7(Jul 1976) pp 79-84

- ROGERS, J.G. "Structured Programming for virtual storage
systems"
IBM Systems Journal 14,4(1975) pp 385-406

- VAN GELDER, A. "Structured Programming in Cobol: An Approach
for Application Programmers"
Comm.ACM 20,1(Jan 1977) pp 2-12

See also references [17] and [18]

E) Impact on Software Production

- BAKER, F.T. "Chief programmer team management of production programming"
  IBM Systems Journal 11,1(1972) pp 56-73

- COOKE, L.H.jr "The Chief Programmer Team Administrator"
  Datamation 22,6(Jun 1976) pp 85-86

- FAGAN, M.E. "Design and code inspections to reduce errors in program development"
  IBM Systems Journal 15,3(1976) pp 182-211

- LUCAS, H.C.jr; KAPLAN, R.B. "A Structured Programming Experiment"
  Computer Journal 19,2(May 1976) pp 136-138

- PETERS, L.J. "Managing the Transition to Structured Programming"
  Datamation 21,5(May 1975) pp 89-96

- YOURDON, E. "Making the Move to Structured Programming"
  Datamation 21,6(Jun 1975) pp 52-56

See also reference [13]