Proceedings of a Codasyl Programming Language Committee Symposium on Structured Programming in COBOL — Future and Present, Henry P Stevenson ed, pages 22-53, Los Angeles California, April 7, 1975

Designing and Coding Program Structures

Michael Jackson
Michael Jackson Systems Limited
5 Scot Grove
Pinner Middlesex HA5 4RT
England

Abstract

The most important question in Structured Programming is the design question: for a given problem, what should be the structure of the program?

This paper presents a design technique which assures a true correspondence between the problem and program structures. The resulting programs are therefore easy both to understand and to maintain. Basic structures are built from sequences, iterations and selections. In the presence of certain problem features the basic technique must be modified to allow backtracking and to allow composition of incompatible structures into a single program.

COBOL coding is shown which correctly implements the recommended structures. Inelegance in the coding should not be a reason for distorting program structure: it may suggest some directions for improvement of the COBOL language.

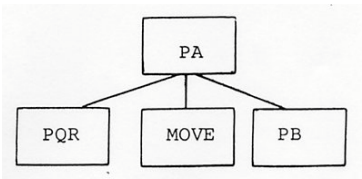
1. Introduction

It is a fundamental tenet, now widely accepted, of Structured Programming that program control structures should be composed of sequences, iterations and selections.

A sequence component has two or more parts. The parts are executed once each, in a pre-specified order, each time that the sequence itself is executed. An example of a sequence component is the COBOL paragraph PA:

```
PA. CALL 'PQR' USING A, B, C.
    MOVE 2 TO D.
    PERFORM PB.
```

The parts of PA are the subprogram PQR, the MOVE operation and the paragraph (or section) PB. We may represent this sequence diagrammatically as:

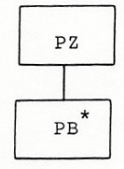


In the diagram the order of execution is shown from left to right. Note that the diagram specifies only the structure of the sequence, its name and the names of its parts: it does not specify the manner of coding.

An iteration component has one part only, which is executed zero or more times. An example of an iteration is the COBOL statement:

```
PERFORM PB UNTIL X = 0.
```

We may represent this iteration diagrammatically as:

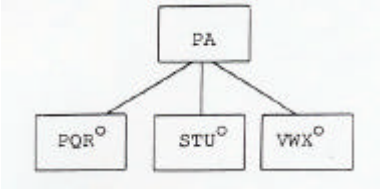


The asterisk on the component PB indicates that PB is to be executed zero or more times — that is, that PZ is an iteration. Note that the component PZ must be named in the diagram, although it is not named in the COBOL coding.

A selection component has two or more parts of which one, and only one, is selected for execution each time the selection itself is executed. An example of a selection is the COBOL section PA:

```
PA SECTION.
PASLCT. IF X IS LESS THAN 5 GO TO PAOR1.
    CALL 'PQR' USING A, B, C.
    GO TO PAEND.
PAOR1.  IF X = 5 GO TO PAOR2.
    CALL 'STU' USING A, B, C. GO TO PAEND.
PAOR2.
    CALL 'VWX' USING A, B, C.
PAEND. EXIT.
```

We may represent this selection diagrammatically as:



In the diagram the ordering of the parts, from left to right, indicates the order in which the selection is to be made. Note that the COBOL coding shown, although inelegant, does correctly implement the selection. We will have need of such coding later.

The reason for restricting ourselves to these three component types is the reason given by Dijkstra [1]:

> "... then we should restrict ourselves in all humility to the most systematic sequencing mechanisms, ensuring that "progress through the computation" is mapped on "progress through the text" in the most straightforward manner."

The three component types are simple to understand. We may therefore expect that by reading the program text we will be able to follow directly the course of its execution.

But there is a further, crucial step to be taken. We must be able to relate the course of the program execution directly to the solving of the given problem, and we will be able to do so only if the program structure reflects the problem structure. The central question of program design is how to achieve this correspondence of structures. If the correspondence fails we will suffer the penalty of difficult maintenance: a small and trivial change to the specification requires a large and diffuse change to the program. A transparent program structure will not help us if it is different from the problem structure.

2. Basic Design Technique

Various design approaches have been suggested. They all attempt to tackle the design task by a direct onslaught. In step-wise refinement the starting point is an abstract program which, if it could be implemented, would solve the whole problem: subsequent steps involve refining the statements of this abstract program into further programs and the statements of those programs into further programs still. In functional decomposition the starting point is a dissection of the whole problem into a number of functions, each of which may then be decomposed in turn. In programming by action clusters the starting point is the recognition of clusters of associated actions which guarantee that specific requirements of the problem will always be met. Broadly, we may characterise the first two approaches as top-down, while the third is bottom-up. But all of them tackle the problem directly by considering the functional specification.

The main thesis of this paper is that such an approach is too difficult: it presents too many choices and too few criteria of the correctness of the choices made. Consider, for example, the problem of processing a simple serial file in which detail records are arranged in groups. A top-down approach might suggest the following program:

```
P: while still more records
   do if change of group-id
      then process group-id change;
      process record;
   end;
```

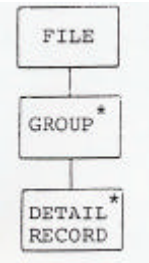
Or consider another example: two arrays, each terminated by a special element 'code', are to be merged into a single array without duplicates. A top-down approach may suggest the program:

```
M: while more elements in array-1 or array-2
   do while next element should come from array-1
      do if not duplicate then move to output from array-1;
         while next element should come from array-2
         do if not duplicate
            then move to output from array-2;
         end;
      end;
   end;
   move 'code' to output;
```

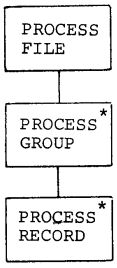
Are these good programs? Will they be easy to maintain? Obviously some interesting arguments may be conducted on these questions. But we seek objective criteria rather than subjective judgements. We want as few as possible of our design choices to be matters of opinion.

We have already resolved to construct program control logic only from sequences, iterations and selections. Our program structures will be hierarchies, in which each component, unless it is an elementary operation (such as MOVE or ADD), will be either a sequence, an iteration or a selection. We now observe that these same component types, and hierarchical structures built from them, may be equally well applied to data as to programs. This is a crucial point: we will base our program structures on the structures of the data they process.

Consider the example of the simple serial file of detail records. The structure of this file is:



We must therefore build a program structure to match:



That is, the basic form of the program must be:

```
P: while still more groups
   do process group-start;
      while still more records for this group
      do process record;
      process group-end;
      end
   end;
```

Much elaboration is, of course, needed before we arrive at a full statement of the program structure. Note, too, that nothing has yet been decided about the manner (or, indeed, the language) in which the program will be coded.

There are important underlying reasons for basing program structure on data structure. The essential aim in design is to make the program structure reflect the problem structure. Everything therefore depends on a full understanding of the problem structure, and this-is best obtained from a consideration of data structures. The general idea that data structures are a model of the problem (or, more properly, of the problem environment) will be familiar to anyone who has worked on a database system. But we can go much further than this:

- The problem environment is modelled in the data structures. At any one moment we must have a data structure which is hierarchical: for some problems (where, for example, the model is necessarily a network) different hierarchical structures must be imposed at different times on the same data, either by use of suitable database software or by simpler methods such as sorting.
- The task to be performed by the program must, ultimately, be expressed in elementary operations of the programming language. Some of these operations (input and output) are needed so that the data structures can be explored in the appropriate way; others (arithmetic and data manipulation) are needed to carry out directly the requirements of the task itself.
- Both kinds of task are naturally associated with components of the data structures. If the program structure contains components properly corresponding to the data components, it will inevitably contain a correct place for each elementary operation. For example, if we have an operation 'add amount in debit transaction to customer balance', we must clearly allocate that operation to a program component which will be executed once per debit transaction and which is itself a part of a component which will be executed once per customer.

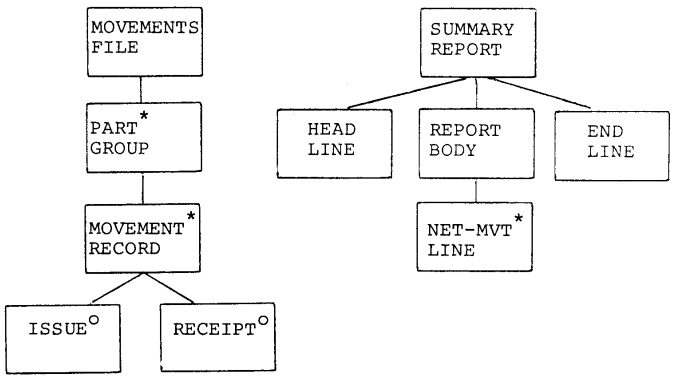
We therefore arrive at the following basic design procedure:

1. Define the data structures
2. Create the program structure from the data structures
3. Express the program task in elementary executable operations
4. Allocate each operation to an appropriate component of the program structure.

This basic procedure is illustrated by a simple example which follows.

3. A Simple Example

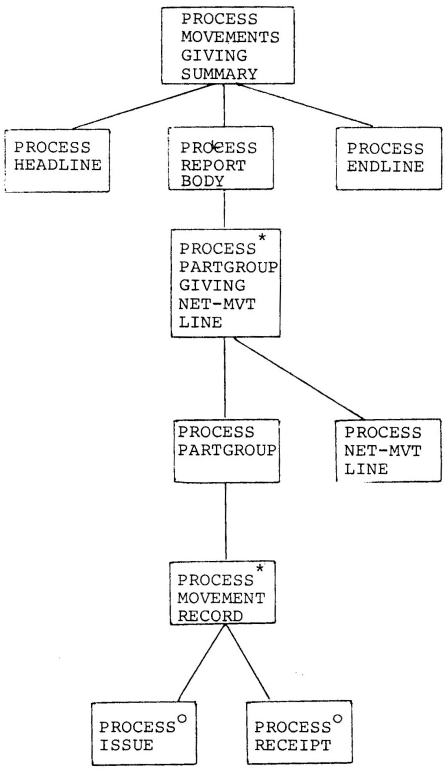
A serial file contains records of issues and receipts of parts in a factory store: the records have been sorted into ascending order by part-number. A summary is to be produced showing the net movement for each part; the summary has a simple heading line, and an ending line showing the number of parts for which movements have occurred in the period of the summary. The data structures are evidently:



We must create a program structure which matches both of these data structures simultaneously. To do so, we look for 1-1 correspondences between components of the two structures. Clearly we have the following correspondences:

- MOVEMENTS FILE corresponds to SUMMARY REPORT. There is one of each, and the report is produced from the file.
- PART GROUP corresponds to NET-MVT LINE. There is the same number of each, and they are ordered so that they correspond pairwise. (The purpose of sorting the file was to achieve this correspondence).

The resulting program structure is:

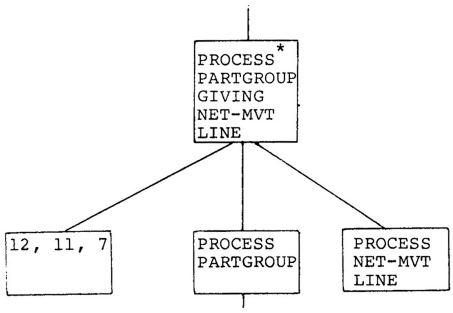


The next step in the procedure is to list the executable operations necessary to carry out the program task. We arrive at the following list:

```
1 open movements file
2 close movements file
3 read movements file
4 display headline
5 display endline
6 display net-mvt line
7 move 0 to net-mvt
8 add receipt-quantity to net-mvt
9 subtract issue-quantity from net-mvt
10 move 0 to num-parts
11 add 1 to num-parts
12 store part-no of group
13 stop run
```

Obviously, the list is largely dependent on the facilities available in the programming language and on certain choices concerning data representation. In particular, we have chosen to use DISPLAY statements for the summary report: hence we need not open or close it.

The next step is to allocate each operation to an appropriate component of the program structure. Sometimes this will require the addition of components to the basic structure, but the allocation should always be trivially easy: any difficulty in allocating an operation means that the program structure is inadequate and must be reconsidered. It will be found convenient to write the operation numbers on the program structure diagram, adding components where necessary. For example, operation 12 (store part-no of group) must be executed once per part group, at the beginning of the part group; so too must operation 11 (add 1 to num-parts) and operation 7 (move 0 to net-mvt). We therefore elaborate the diagram:



and similarly for the other operations.

We are now ready to choose suitable implementations for the components. A possible set of choices gives the following program.

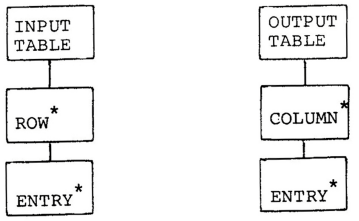
```
DATA DIVISION.
FILE SECTION.
FD MVTS ... .
01 MVTREC.
   02 CODE PIC X.
     88 ISSUE VALUE 'I'.
     88 RECEIPT VALUE 'R'.
   02 PARTNO PIC X(7).
   02 QTY PIC 9(6) COMP.
WORKING-STORAGE SECTION.
77 NETMVT PIC S9(6) COMP.
77 NUMPARTS PIC 9(4) COMP.
01 STOREDKEY.
   02 PARTKEY.
   03 EOF PIC X.
   03 PARTNO PIC X(7).
01 RECKEY.
   02 PARTKEY.
     03 EOF PIC X.
       88 ENDFILE VALUE 'E'.
       88 NOTENDFILE VALUE SPACE.
     03 PARTNO PIC X(7).
PROCEDURE DIVISION.
P-MVTS-SUMMARY.
    OPEN INPUT MVTS. MOVE SPACE TO EOF IN RECKEY. PERFORM P-READ-MVTS.
    MOVE 0 TO NUMPARTS. DISPLAY 'SUMMARY REPORT'.
    PERFORM P-PARTGP-NET-MVT-LINE UNTIL ENDFILE.
    DISPLAY NUMPARTS, ' PARTS AFFECTED'.
    CLOSE MVTS. STOP RUN.
P-PARTGP-NETMVT-LINE.
    MOVE RECKEY TO STOREDKEY. ADD 1 TO NUMPARTS.
    MOVE 0 TO NETMVT.
    PERFORM P-MVT-REC UNTIL PARTKEY IN RECKEY NOT = PARTKEY IN STOREDKEY.
    DISPLAY 'PART-NO ', PARTNO IN STOREDKEY,
            ' NET-MOVEMENT ', NETMVT.
P-MVT-REC.
    IF ISSUE SUBTRACT QTY FROM NETMVT ELSE ADD QTY TO NETMVT.
    PERFORM P-READ-MVTS.
P-READ-MVTS.
    READ MVTS AT END MOVE 'E' TO EOF IN RECKEY.
    IF NOTENDFILE MOVE PARTNO IN MVTREC TO PARTNO IN RECKEY.
```

Note that the allocation of the operation ‘read movements file’ follows a general principle for reading serial input: we ensure that the next record is always available for inspection by reading the first record when the file is opened and reading subsequent records at the end of each program component which completes the processing of a record.

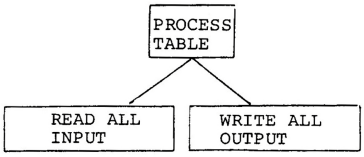
4. Structure Clashes

A very large number of data processing problems can be successfully handled using only the basic technique described above. Sometimes, however, the data structures cannot be fitted together so easily to form a program structure: the required 1-1 correspondences cannot be found.

An obvious example of such a clash is a difference in order between input and output. A program is required to print out a table from a deck of cards: each card contains one row of the table, but the table is to be output by columns. We have the two data structures:

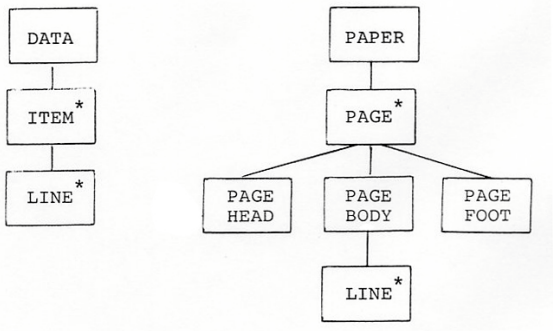


There is a good 1-1 correspondence between INPUT TABLE and OUTPUT TABLE, but none elsewhere. The solution to this clash is well known: if enough main storage is available to hold the complete table we may structure the program as:



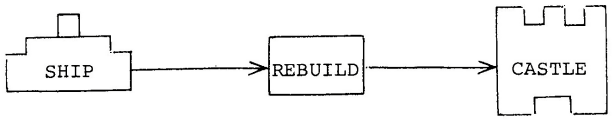
while in the absence of sufficient main storage we may sort the entries from row order into column order. In either case recognise the impossibility of superimposing the two data structures one on another.

Another example of a structure clash arises frequently when preprinted stationery is used for variable-length output. Suppose that the output consists of a number of items, where an item may occupy as few as two lines or as many as 100; suppose also that this output is to be printed on pages of fixed size (say, 60 lines), each page having a heading and footing. Items may be split between pages, and the same page may contain several items. The data structures involved are:

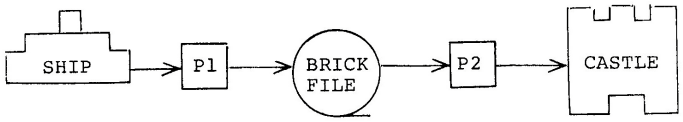


There is no way of fitting together the components ITEM and PAGE in a single program structure. Neither directly nor indirectly are they related as parts of a sequence or a selection; nor is one an iteration of the other, because a page may contain several items and an item may occupy several pages. There is therefore a structure clash which we ignore at our peril.

The solution to a structure clash always involves abandoning any attempt to form a single program structure: instead we content ourselves with creating two or more programs mediated by serial files. Consider, as a further illustration, the problem of dismantling and rebuilding a LEGO model. We have a ship, but we want a castle: we know that the ship and castle both require the same number of bricks.

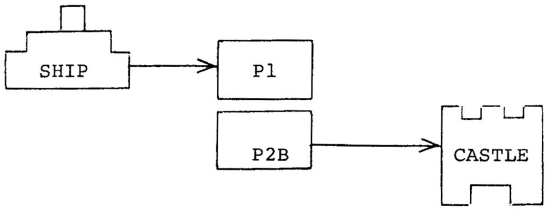


We may reasonably suppose that there is a structure clash here: the ship consists of a hull, a superstructure and a funnel, while the castle has a very different structure. The solution to the clash is to create two programs, one to deal with each structure:



Pl dismantles the ship into bricks: each record of the intermediate file is a brick. P2 builds the bricks into the castle. Pl therefore need know only about the ship structure, while P2 need know only about the castle. We choose bricks as the records of the intermediate file because a brick is the highest-level component for which we have a 1-1 correspondence between the two structures.

Of course, the solution is inefficient. We have doubled the elapsed time to run the program, because P2 cannot start to build the castle until Pl has finished dismantling the ship. However, we can eliminate the intermediate file and, with it, the inefficiency. We can design the system as shown, but code (not design) one of the programs Pl and P2 as a subprogram of the other. For instance, we can code P2 as a subprogram, P2B, of Pl:



Pl will call P2B whenever Pl wants to dispose of a brick; the function of P2B is to take the given brick and add it to the castle.

It is essential to recognise that the design of P2B is the same thing as the design of P2, and that the only difference is in the coding. Where P2 contains a 'read brick' statement, P2B must contain coding which has the following effect:

- remember which point has been reached in processing the brick file and building the castle: this is a unique point in the program text, immediately following the read statement;
- return to the calling program (in this case, Pl) to obtain the next brick record;
- resume operation of P2B, when it is next called, at the remembered point.

A suitable implementation in COBOL is the following:

- Code each read statement as
  ```
  MOVE n TO'QS. GOBACK. Qn.
  ```
  where the read statement is the *n*th in the program text. For the 1st read statement, which must appear at the beginning of the program, only the paragraph name Q1 is coded.
- Define a level-77 item QS in working-storage with the initial value of 1.
- At the entry point of P2B code
  ```
  ENTRY 'P2B' USING BRICK-RECORD.
  GO TO Ql, Q2, ... Qx DEPENDING ON QS.
  ```
  where there are *x* read statements in the program text.
- In place of the STOP RUN statement of P2 code
  ```
  GOBACK.
  ```

It will be apparent that such a mechanism cannot be relied on if any of the read statements is within the scope of a PERFORM or CALL statement in P2. It is necessary, therefore, to code much of the program in a 'flat' style, relying entirely on conditional and unconditional GO TO statements for the

control logic. The inelegance of such coding is obvious. But to remove it would require far-reaching changes in the language and its run-time environment.

The coding technique described may be called 'program inversion': we 'invert P with respect to the bricks file' to obtain the subprogram P2B. The importance of the technique is hard to exaggerate. It provides a major simplification of the design task, because we need never design anything other than a serial file processing program. It unifies system and program design, because it allows us to view a system as a very large program involving several structure clashes. It solves problems of synchronisation, because it allows us to design a program whose structure reflects a complete time sequence and then invoke that program once per element of that sequence.

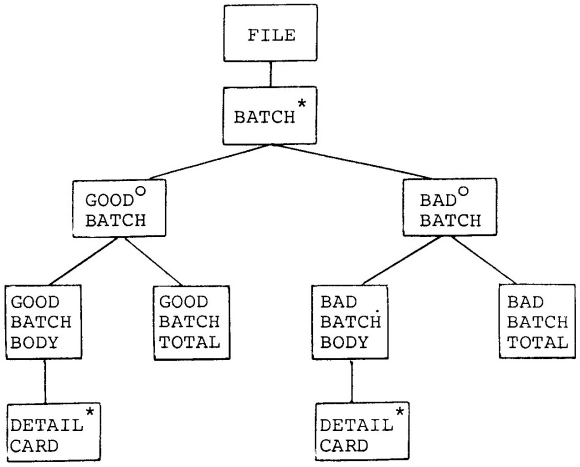
We give three examples of the power of the program inversion technique.

1. In a conversational on-line system, the processing of one conversation is a simple serial program. The input file is the file of messages keyed in by the user; the output file is the file of responses displayed at the terminal. Inverting the program with respect to the input file gives a transaction processing routine which can be invoked to process one input message. The state vector (ie the variable QS and other variables which must persist from one invocation to the next) is held in an area associated with the conversation (eg the SPA in an IMS system).
2. An interrupt handler may be thought of as processing a serial input file of interrupt records. If there are *n* currently active processes which may cause interrupts, the handler separates the single input file into *n* files, to be processed, in general, by *n* programs. Inverting each of these with respect to its own input file, and the original handler with respect to the serial file of interrupts, we arrive at a suitable structure. Usually fewer than *n* distinct programs are required, because the active processes are similar or identical: where the active processes are identical, only one program is required, with n state vectors. The interrupt handler is responsible for identifying and accessing the appropriate state vector.
3. A system is required to process insurance policies. The lifetime of a policy may be as much as a hundred years, but the policy file is to be updated daily. The core of the system is a program which runs for a hundred years, processing all of the input for one policy only. Inverting this program with respect to its input file of transactions, we obtain a program which may be invoked to process a single transaction for the designated policy. By keeping several state vectors for this program (conventionally called a 'file of policy records') we can arrange to process as many policies as we wish, and to process them as often as transactions are available. Depending on the requirements and on the software environment we may construct a daily batch system or an on-line system without affecting the transaction processing program which is at the core of the system.

5. Backtracking

In the examples discussed so far, we can find our way about the serial input files by reading one record ahead. For instance, in the stores movements problem, we know when there are no more part groups — the next record is the end-of-file record; and we know when there are no more movements for the current part — the next record is end-of-file or else it has a different part number.

Sometimes we will need to read more than one record ahead. More significantly, we are sometimes unable to find our way around the data structures by reading any predetermined number of records ahead. The following data structure occurs commonly:



However many cards we plan to read ahead, there may yet be a batch containing too many detail cards to be handled in this way. And yet the above structure is the natural and right structure for the data and hence for the program. Its only difficulty is that we cannot see into the future and therefore cannot decide, when we start to execute the component PROCESS BATCH, whether we have a good or a bad batch.

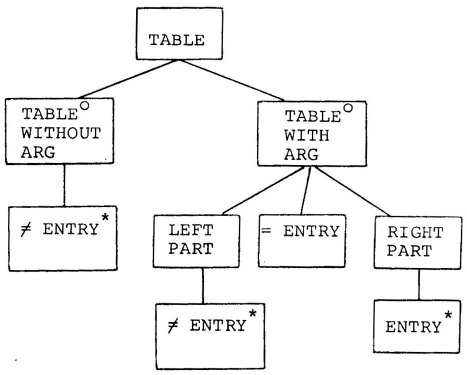
An interesting example of a backtracking problem, which we will use to illustrate the necessary design technique, is the familiar serial table look-up. If the argument is present in the table the associated value is to be returned as result, otherwise an error routine is to be executed.

Evidently, the problem is structured as a selection; so too is the output data: it is either the required value or an error message. The table itself either contains, or does not contain, the argument sought. But we cannot know in advance whether the argument is in the table.

We use the following three-stage technique for solving backtracking problems:

- Stage l. Structure the problem as a simple selection, ignoring the impossibility of evaluating the condition.
- Stage 2. Recognising that the selection is really an assumption, introduce into the program text the statements which correspond to abandoning the assumption when it has proved untenable. These are 'quit' statements, implemented as transfers of control to the second part (the else-clause) of the selection.
- Stage 3. Consider and deal with the side-effects which result from partial execution of the now abandoned part of the selection. Side-effects may be dealt with *en masse*, by saving and restoring the relevant parts of the state of the computation. Or they may be dealt with piecemeal, distinguishing intolerable, neutral and beneficent side-effects.

For our Stage 1 solution we use the following data structure for the table:



The correspondences between input and output data structures are trivial: clearly TABLE WITHOUT ARG corresponds to the error message, and TABLE WITH ARG corresponds to the required value. Using the 'flat' style of coding, we arrive at the following program:

```
P-LOOK-UP-SEQ.
    MOVE 1 TO S.
P-TABLE-SLCT.
    IF ARGUMENT-IS-IN-TABLE GO TO P-TABLE-OR.
    PERFORM ERROR-ROUTINE.
    GO TO P-TABLE-END.
P-TABLE-OR.
P-LEFT-PART-ITER.
    IF ENTRY-KEY (S) = ARG GO TO P-LEFT-PART-END.
    ADD 1 TO S.
    GO TO P-LEFT-PART-ITER.
P-LEFT-PART-END.
    MOVE ENTRY-VALUE (S) TO RESULT.
P-TABLE-END.
P-LOOK-UP-END.
```

The only defect of this program is that we have no way of evaluating the condition ARGUMENT-IS-IN-TABLE. This defect is remedied by the next stage.

For Stage 2, we ask ourselves what could invalidate the assumption that the argument is missing from the table. Obviously, finding an equal entry in the component TABLE WITHOUT ARG. So we must examine all of the entries in TABLE WITHOUT ARG, and 'quit' our assumption if one of them turns out to be equal to the argument. The changes required for Stage 2 give the following program:

```
P-LOOK-UP-SEQ.
    MOVE 1 TO S.
P-TABLE-ASSUME.
P-TABLE-WITHOUT-ITER.
    IF S GREATER THAN MAX GO TO P-TABLE-WITHOUT-END.
    IF ENTRY-KEY (S) = ARG GO TO P-TABLE-ADMIT.
    NOTE QUIT IF ENTRY =.
    ADD 1 TO S.
    GO TO P-TABLE-WITHOUT-ITER.
P-TABLE-WITHOUT-END.
    PERFORM ERROR-ROUTINE.
    GO TO P-TABLE-END.
P-TABLE-ADMIT.
P-LEFT-PART-ITER.
    IF ENTRY-KEY (S) = ARG GO TO P-LEFT-PART-END.
    ADD 1 TO S.
    GO TO P-LEFT-PART-ITER.
P-LEFT-PART-END.
    MOVE ENTRY-VALUE (S) TO RESULT.
P-TABLE-END.
P-LOOK-UP-END.
```

For Stage 3 we need to consider the side-effects. The only significant side-effect is the incrementing of S in the iteration P-TABLE-WITHOUT-ITER. This is clearly a beneficent side-effect: it leaves S pointing at the equal entry, and so makes the iteration P-LEFT-PART-ITER redundant. So the final program is:

```
P-LOOK-UP-SEQ.
    MOVE 1 TO S.
P-TABLE-ASSUME.
P-TABLE-WITHOUT-ITER.
    IF S GREATER THAN MAX GO TO P-TABLE-WITHOUT-END.
    IF ENTRY-KEY (S) = ARG GO TO P-TABLE-ADMIT.
    NOTE QUIT IF ENTRY =.
    ADD 1 TO S. GO TO P-TABLE-WITHOUT-ITER.
P-TABLE-WITHOUT-END.
    PERFORM ERROR-ROUTINE. GO TO P-TABLE-END.
P-TABLE-ADMIT.
    MOVE ENTRY-VALUE (S) TO RESULT.
P-TABLE-END.
P-LOOK-UP-END.
```

6. Designing and Coding

In some circles Structured Programming is synonymous with the avoidance of GO TO statements: the quality of a program is held to be in inverse proportion to the number of GO TO statements it

contains. In such circles some of the coding shown in this paper will be thought not merely inelegant but heinously criminal.

I believe that this is a very serious mistake. And it is a mistake that the computing community has made once before. When Modular Programming was introduced to the general body of installations it was heralded as a panacea for all the aches and pains of programming: all you had to do was to break your program down into separately compiled modules, and magically it would become easy to understand, to test and to maintain. The promise was scarcely fulfilled. Programmers who had written bad monolithic programs now wrote bad modular programs. The cost of integrating the modules into a program often outweighed the savings made in their creation. Maintenance was often harder, not easier, because twenty modules now had to be changed and recompiled where before only one monolithic program was affected.

The crucial factor, of course, was the quality of program design. Modular programming benefited those who could use it to support good design technique. I believe that we have a similar situation today with Structured Programming. We are in danger of confusing the external appearance of a program with its internal health; of believing that a program without a GO TO must be a good, well-structured program, which will be easy to understand and maintain.

The underlying design is much more important. Good design can be coded in any language which provides conditional and unconditional GO TO statements and program labels. And if the language does not provide the constructs we need to code our design elegantly, then we must resolve to code it inelegantly. What we must never do is to allow the vagaries of the language to dictate program structure.

Reference

[1] O-J Dahl, E W Dijkstra & C A R Hoare; *Structured Programming*; Academic Press. 1972. pp 22-23.