MORE CONSTRUCTIVE METHODS OF PROGRAM DESIGN

M.A. Jackson

Rapporteurs: Mr. M. King
Mr. M. Martin
Mr. D. McGlade

Abstract

A fundamental consideration in design is structure. Program structure should be based on data structure for proper correspondence between program and problem. Programs can be decomposed into 'simple programs' implementable in ordinary languages. Real world entities are modelled by processes of the same lifetime. Possible implementations of systems are suggested. The experience of teaching shows that optimisation should be avoided at lower levels, and that implementation and modelling be kept separate at higher levels.

PART I

Consider first the general question of what is meant by design. It could be said that it is the activity of decomposition; of recognising that the whole has parts and of saying what those parts are and how they should be put together.

Suppose that one is to construct an object, which by some estimate is to have N parts, each part being chosen from P different part types. Imagine, for example, writing a program of 1000 statements, making N=1000. The choice of statements will be from all the statements of the programming language and so P is between 10 and 1000. The total number, P^N, of all such programs is thus too large to think of designing the program by simply putting together individual statements.

It is clear that some kind of hierarchical structure, for example, hierarchies of procedures, levels of abstraction or steps of refinement effectively reduces N since no object has more than, say, five parts. 'Modular programming' certainly aims to offer these benefits, but using a hierarchy of modules to reduce N alone does not help if the number of different modules is possibly infinite.

Structured programming (GO TO statement considered harmful and so on) reduces P since only sequences, conditional constructs, iterative constructs and elementary objects of the programming language are allowed. This is the benefit to be gained from adopting structured coding but in practice it is still possible to have a large number of apparently attractive designs. This situation suggests that a more constructive approach to designing structured programs is required. In the methodology proposed here a single specific part type is put forward as a basis for design.

Consider a simple example. There is a deck of cards which record some information, stock movements in an inventory system, for example. These have been sorted so that all the details for each stock item have been brought together and it is required to make a summary report of the net movement for each item.

In the proposed methodology the first thing to do is to write down the structure of the data objects involved. This may be in the form of a diagram, as in Figure 1, or any other suitable way.

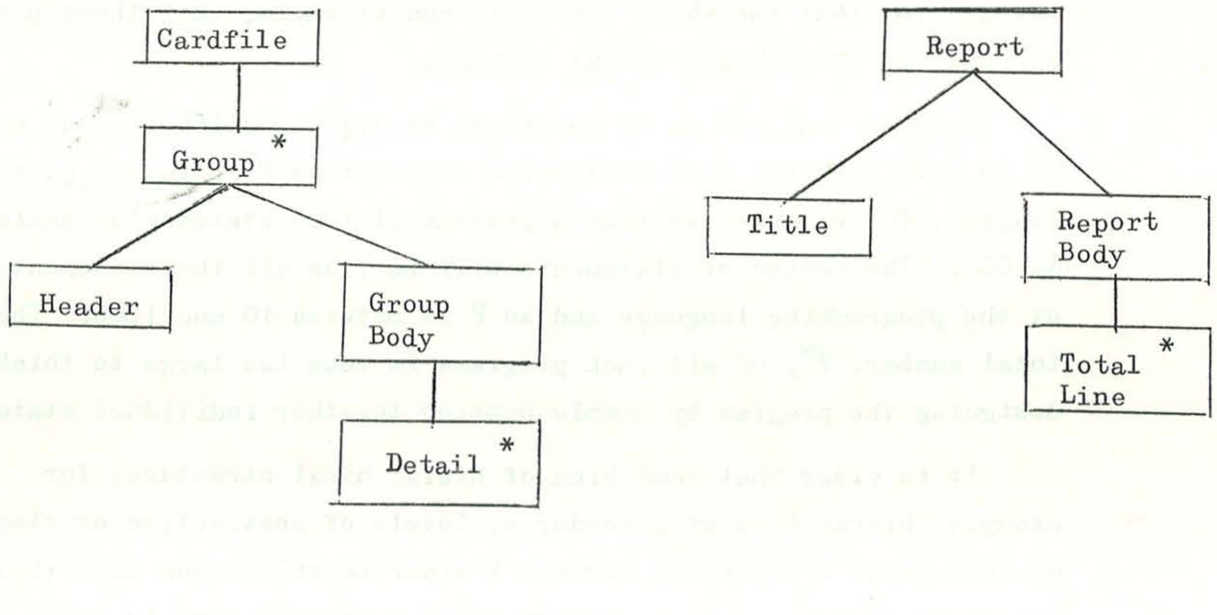


Figure 1

It can be seen from Figure 1 that a card file consists of zero or more groups (an asterisk in a box indicates an iteration of zero or more of the items so marked). Each group is made up of a header card followed by a group body which in turn is composed of zero or more detail cards. The output report is to consist of a title followed by a report body composed of zero or more total lines.

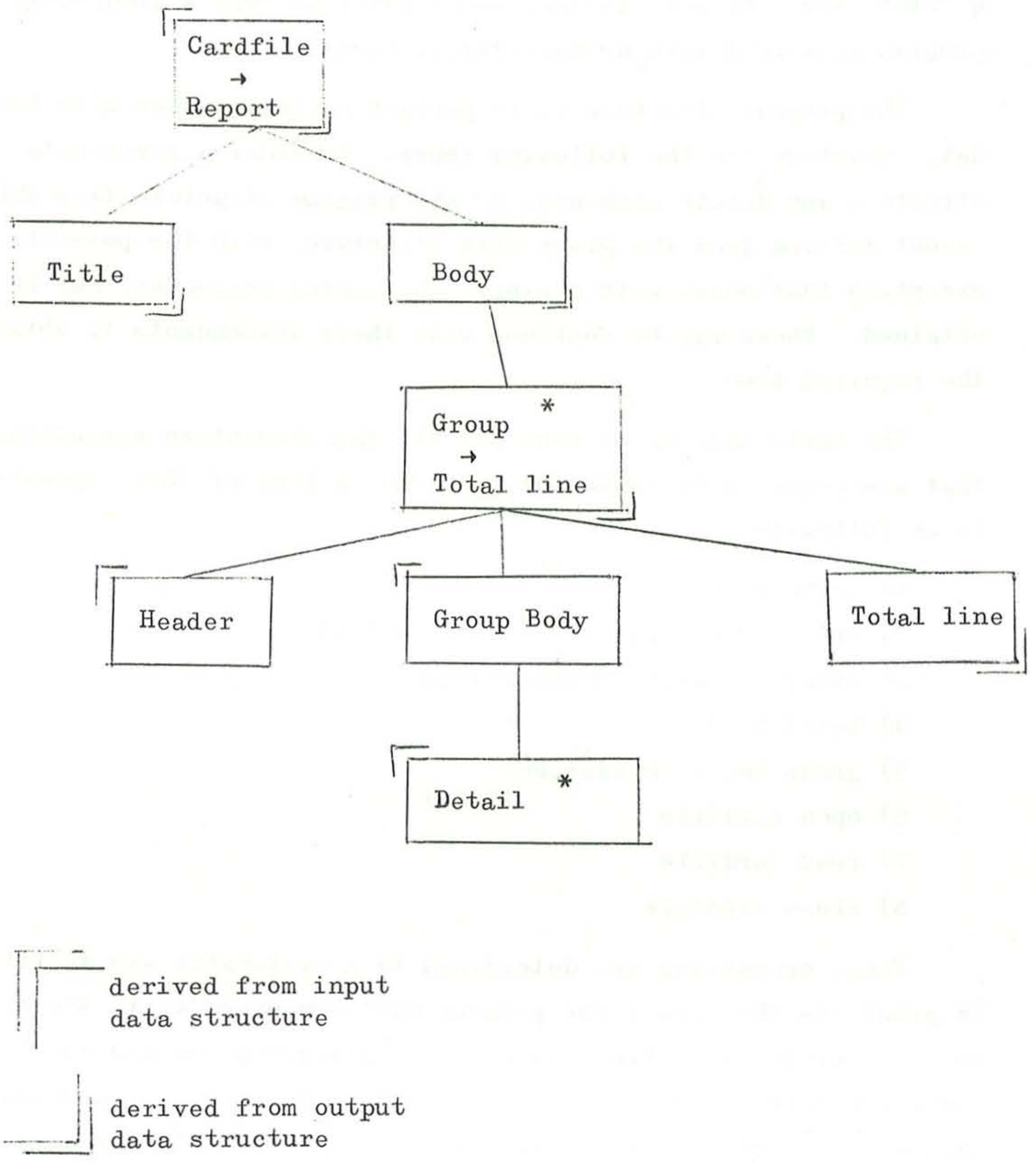


Figure 2

The next step is to deduce the program structure from the data structures. Since the program must operate on this data it must in some sense be identical to those structures. Such a program is given in Figure 2. This indicates that in order to produce a report from a card file the program must first produce a title and then a report body. To produce the report body it must read a group of cards and write a total line zero or more times. To do this the program must read the group header and the group body before writing a total line. Finally at the lowest level to read a group body the program must read zero or more detail cards.

The program structure is in perfect correspondence with the data structures in the following sense. Consider a given data structure and delete each node of the program structure tree which is not derived from the given data structure, with the possible exception that nodes with a single uniterated descendant may be obtained. These may be combined with their descendants to obtain the required tree.

The next stage is to consider all the elementary operations that are required to solve the problem. A list of these operations is as follows:-

1) write title
2) write total line (group key, total)
3) total := total + detail.amount
4) total := 0
5) group key := header.key
6) open cardfile
7) read cardfile
8) close cardfile

These operations are determined in a systematic way as follows. In producing the report the program must somewhere write the title and also write total lines consisting of a group key and total. To obtain this total the amount of each stock movement on each detail card must be added to the total, the total must be initialised and so on. Only lists of elementary operations of a modest size are ever required for even the largest programs.

The next step is to consider how these operations will fit into the program structure. It is vital to find out at the earliest possible moment whether the program structure is correct, and this is answered by the ease with which these elementary operations fit the program structure. For example, consider a program structure which at the top level resembles Figure 3. Here the program is to be an iteration of processing cards. Now compare the ways in which the operation of writing a total line will fit the two program structures. In Figure 2 it clearly corresponds to the box which says produce a total line. In Figure 3, however, some kind of complex conditional is required to incorporate that operation into the program. From this one concludes that the program structure of Figure 3 is wrong, and it is wrong because the program structure is not in perfect correspondence with each of the data structures. Moreover, similar difficulties will be encountered with any other program structure for which this correspondence does not hold.

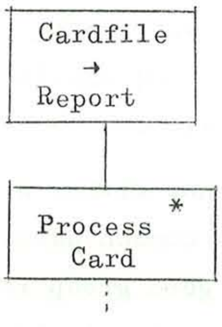


Figure 3

Finally the program given in Figure 4 is obtained by an obvious transcription of Figure 2 with the elementary operations included in their proper places. Here seq - end encloses a compound group of sequential operations, and iter - end encloses a group of operations to be iterated under the control of the given condition. At this point the solution is essentially complete, but in practice it may be necessary to transform this text, either mechanically or by hand, into some language which can be run on a machine.

```
CARDFILE - REPORT   seq
                      open cardfile; read cardfile; write title;
REPORT - BODY         iter until cardfile.eof
                          total:=0; groupkey:=header.key; read cardfile;
GROUP - BODY              iter until cardfile.eof | card.key ≠ group key
                              total:=total+detail.amount; read cardfile
GROUP - BODY              end;
                          write total line (group key, total)
REPORT - BODY         end;
                      close cardfile
CARDFILE - REPORT   end
```

Figure 4

Since the program structure derived in this way closely reflects the data structures, great care must be taken to ensure that the input data structure truly defines the expected input. This is because these data structures define a valid set of input and if any input does not have this structure then the program will produce unspecified results. In the real world people make mistakes and so Figure 1 is not really valid, and should be changed to Figure 5 to take account of such possibilities.

In Figure 5 a group is now either a good group or an error group (circles in boxes with common parents indicates mutually exclusive alternatives). A good group is the same as a group in Figure 1 whilst an error group is simply an iteration of cards.

No attempt should be made to define a data structure which takes account of the way in which errors might be handled. This is because the resulting program structure will be more related to the solution of the error handling problem than to its true objectives.

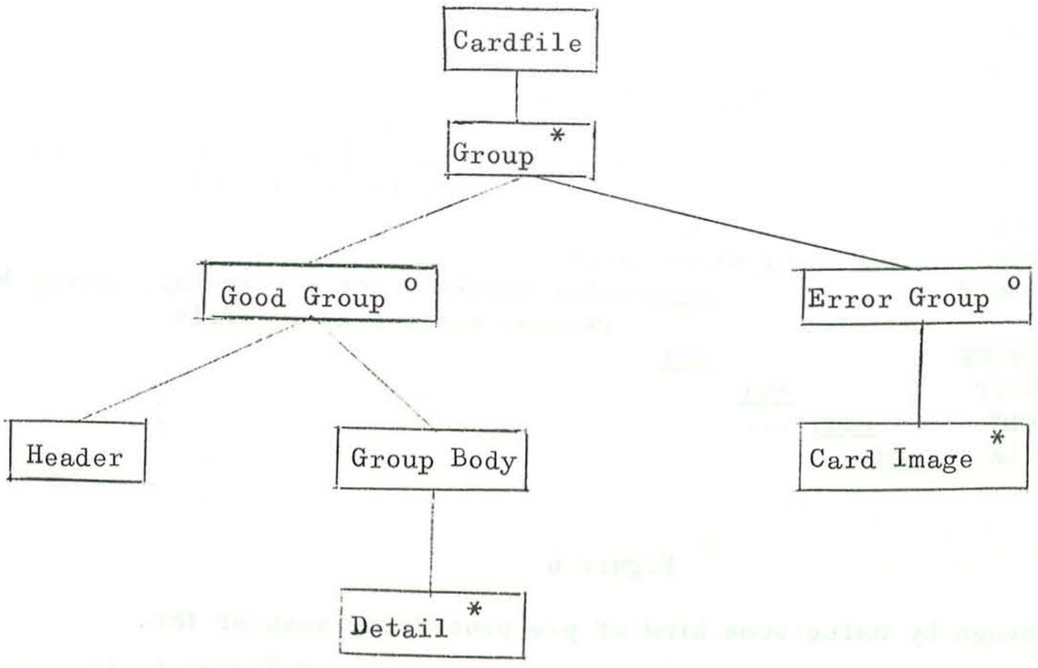


Figure 5

The trouble with the structure given in Figure 5 is that in order to recognise either a good or an error group a potentially infinite amount of look-ahead is required. In order to overcome these difficulties a technique to be called 'backtracking' is introduced and is applied in three stages.

At the first stage the assumption is made that it is possible to determine whether a group is either a good or an error group at any suitable moment. This enables the program to select whether it must process a good or an error group. Such a program is given in Figure 6, where the key words <u>select</u> and <u>or</u> perform such a function.

```
CARDFILE        seq ...
FILE-BODY          iter until cardfile.eof ...
CARD-GROUP             select good group
                             process header; read cardfile;
GROUP-BODY                 iter until cardfile.eof | card.key ≠ group key
                                   process detail; read cardfile
GROUP-BODY                 end
CARD-GROUP             or error group
ERROR-GROUP                iter until cardfile.eof | card.key ≠ group key
                                  process card; read cardfile
ERROR-GROUP                end
CARD-GROUP             end
FILE-BODY         end; ...
CARD FILE       end
```

Figure 6

Although by using some kind of pre-processing scan of the input data it might be possible to use the program of Figure 6, in the second stage this assumption is dropped. Instead the program explicitly assumes that it will process a good group, but as soon as evidence to the contrary is discovered it is prepared to admit that it is processing an error group and take appropriate action. Unfortunately performing some operations on the assumed good group may interfere with the subsequent processing for an error group, so that at stage three all side effects of the initial processing must be repealed. Thus the program given in Figure 7 is finally obtained.

```
CARDFILE        seq ...
FILE-BODY           iter until cardfile.eof
                        posit good group
                             note cardfile;
                             quit CARD-GROUP if ¬ header;
                             process header; read cardfile;
GROUP-BODY                   iter until card group.eof | card.key ≠ group key
                                 quit CARD-GROUP if ¬ detail;
                                  process detail; read cardfile
GROUP-BODY                   end
CARD-GROUP              admit error group
                             restore cardfile;
ERROR-GROUP                  iter until card group.eof | card.key ≠ group key
                                 process card; read cardfile
ERROR-GROUP                  end
CARD-GROUP              end
FILE-BODY           end; ...
CARD FILE       end
```

Figure 7

In Figure 7 posit is used to indicate an assumption. Quit essentially performs a 'go to' the section of code where the program admit's that an error has been made. The operations 'note cardfile' and 'restore cardfile' are primitives of the programming environment which allow the reading position of the cardfile to be backed up by 'restore' to where it was last 'noted'. This is required to allow processing of error groups to proceed normally and is an example of repealing side effects.

The important point about this approach is that the program structure is determined at the first stage, subsequent stages being textual modifications that allow such a structure to be used.

So far the only problems discussed have been those for which a suitable program structure may be derived from the data structures involved. Consider the situation where instead of the cards being read one at a time, they must be read from magnetic tape in blocks of some arbitrary number of card images. The input data structure is now given in Figure 8; it is no longer possible to say anything about the division into groups without conflicting with what we are obliged to say about the division into blocks. Such a situation is called a 'structure clash' and in this case a 'boundary clash' since the boundaries of the blocks are not aligned with the boundaries of the groups.

The solution to this particular problem is immediately obvious and supported by manufacturer's software. In general, the method of overcoming a single boundary clash is to introduce an intermediate file and to split the program into two, as given in Figure 9.

Program PA just deblocks the input and produces an unblocked file x whose structure is that of earlier examples. The file x may then be used as input to the program PB, the solution to earlier problems.

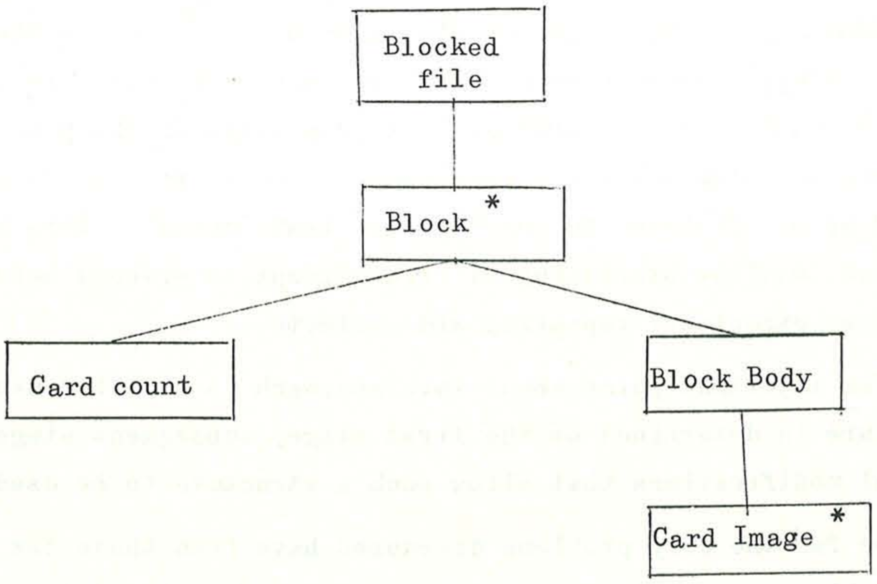


Figure 8

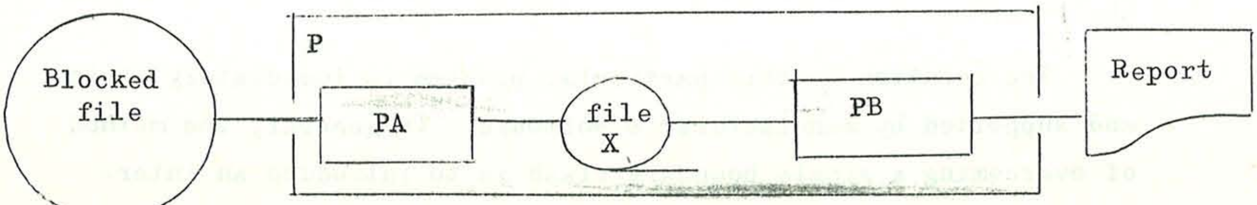


Figure 9

The overall program P is called a 'complex program' because it contains a structure clash. P is decomposed into the 'simple programs' PA and PB which do not have structure clashes. The decomposition of complex programs into simple programs is of primary importance in the design of any non-trivial system.

Such a decomposition is possible in even more complex situations. Suppose that instead of the card file being suitably sorted, it is constructed by several users; each user first puts a header card into the file and then over a period of time adds successive detail cards. Thus groups retain their overall order but are interleaved with cards from other groups. This gives the input data structure of Figure 10.

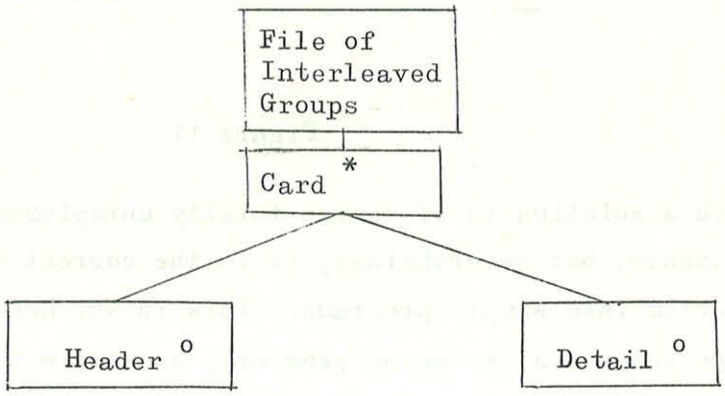


Figure 10

Once again this gives rise to a structure clash, referred to as an 'interleaving clash'. It is resolved by undoing the interleaving in a rather simple-minded way, given in Figure 11. The simple program PA produces a file for each group in the interleaved file. The programs PG1, PG2, ... PGN recombine these files to obtain a file with the same structure as the one in the original problem.

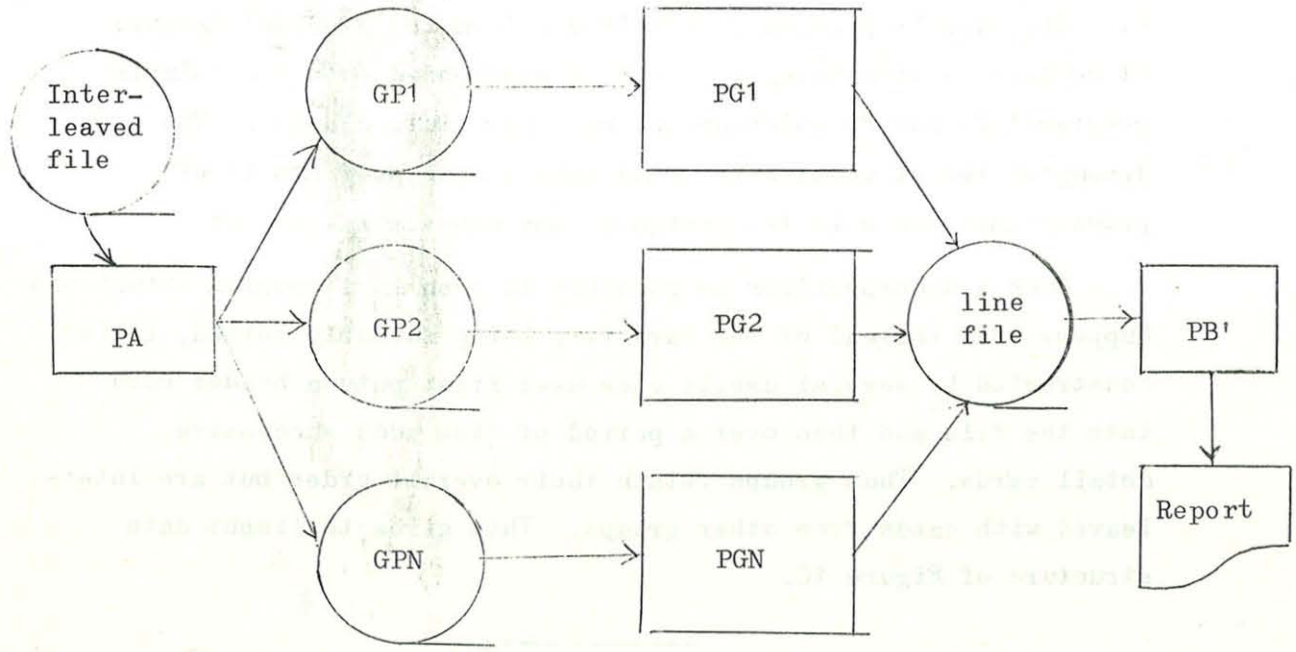


Figure 11

Such a solution is of course totally unimplementable in any literal sense, but nevertheless, it is the correct decomposition of this problem into simple programs. This is the decomposition that should be sought in designing programs, as simple programs are the only 'part types' in this approach to design.

As well as not containing any structure clashes, simple programs have some other important properties. They use serial input and output exclusively, there being no concept of direct access transput within them. Simple programs also observe a standard file protocol. On input the file is opened, at least one read is performed, records being read in advance of any processing, before finally the file is closed. On output the file is opened, zero or more records are written and finally the file is closed.

PART II

Because of the impracticability of the solution to the interleaving clash, physical serial input and output is not always desirable, although possible using disks or drums. An elegant solution might be to use coroutines or classes, but in a commercial or industrial environment languages with such features are not generally available.

The technique of Program Inversion provides a crude implementation of a system in the more common languages, such as COBOL and PL/I. Consider, for example, the small part of a system shown below in Figure 12.

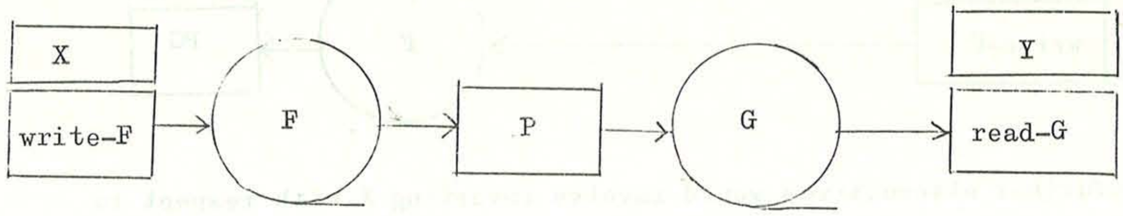


Figure 12

It represents a program X writing to a file F by invoking some procedure or 'access method' write-F. Program P reads from file F and writes to file G, whereas program Y reads from file G via some read procedure read-G. X, P and Y are simple programs. In an actual implementation of this system it might be desirable to reduce the number of tape-drives needed, and an obvious way to do this is (Figure 13) to replace the write-F procedure with a subroutine PF that writes directly to the file G. PF might be described as being a 'plug-compatible replacement for write-F'.

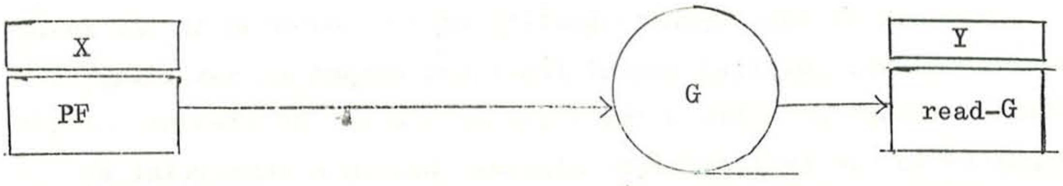
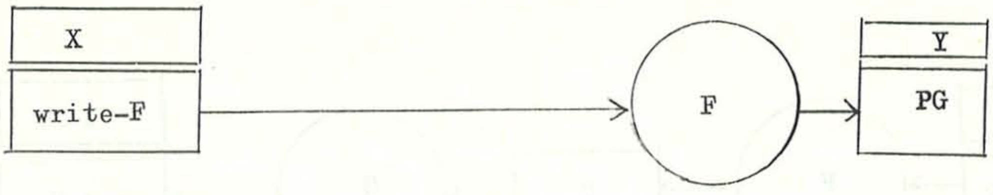


Figure 13

Program X and file G perceive no difference between the situation in Figure 13, and that in Figure 12. PF is termed 'P inverted with respect to the file F'. The symmetry of the situation allows as an alternative P to be inverted with respect to the file G, producing a replacement PG, for the procedure read-G as shown below. Again, Y cannot tell the difference between read-G and PG.



Two further alternatives would involve inverting X with respect to file F producing a procedure that would be called by P and, the symmetrical equivalent, inverting Y with respect to file G.

These different interpretations of the same system show the absurdity of suggesting that top-down and bottom-up are absolute and decidable directions, as some people seem to believe. Which of the programs P, X and Y in the example above would be the top? Clearly, such a small part of a system bears no relation to the fundamental structure of the whole.

Program Inversion can therefore be regarded as a mechanical coding procedure: having written the program P, inversion consists merely of deciding on a particular suitable implementation for read-F and write-G. Were PF to be implemented in PL/I, it might appear as follows:

```
PF: procedure ...
     declare L array of labels(10)static initial(L1, L2, ... , L10);
     declare N integer static initial(1);
     ...
     goto L(N); L1: ... ...
     N:=2; return; L2: /*read F */ ... ...
     N:=3; return; L3: /*read F */ ... ...
```

Initially, X calls PF and since the 'switch variable' N is 1 the code after label L1 is executed until procedure PF wishes to execute a read-F. Instead of a read, however, it merely remembers the text pointer by setting N to (for example) 2, and returns to the calling program X. When X next calls PF, the procedure will continue from Label L2 due to the 'computed goto', having gained from X the information it would have originally been given by a read-F. The method is crude but feasible even in COBOL and PL/I, and removes the need to consider whether X is above or below P, but it has some unfortunate side-effects. An obvious one is the tedium of hand-coding the method. Use of a macro-processor or pre-processor is highly desirable. However, a far worse implication of the technique is that since the run-time stack cannot be preserved after the return to X (just before label L2, for example), the program must ensure that the stack is in fact empty. This means that the code must be 'flattened', with all the desirable control features (such as do-while) converted into the equivalent set of goto's and if's. Here again, a macro or pre-processor is desirable. So, whenever the procedure PF wishes to do the equivalent of a read-F, the values of all the variables, as well as the text pointer, must be preserved. This set of values is called the state-vector, and since the language does not preserve and restore it, the programmer must do so explicitly. It is important in the procedure PF to interpret the read-F points not as in Figure 13 (a subroutine returning to its invoking level), but as in Figure 12, where the read suspends the program P until the completion of the read re-activates it.

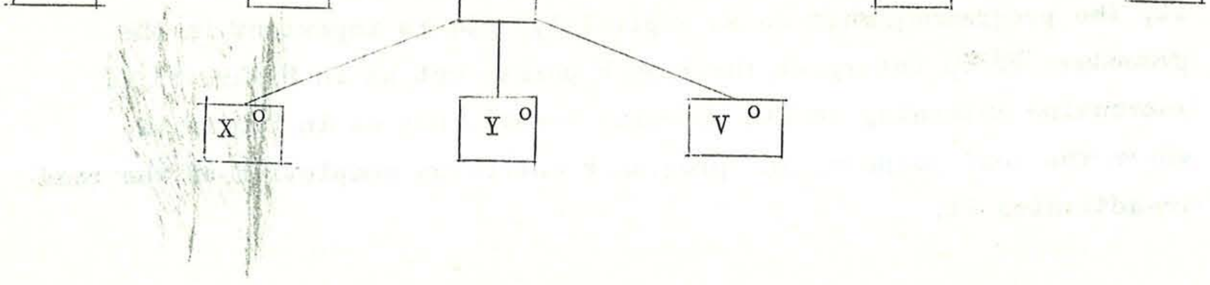
The successful application of Program Inversion to the coroutine oriented problem above is not, however, the whole of the matter. In Data Processing, and apparently many other areas of Computing Science, people are continually setting themselves gratuitous brain teasers typical of those found in the Sunday papers where, for example, one is given an incomplete set of relationships between members of a family, and asked to determine which two members are brothers. This very difficult teaser has an obvious solution once the family tree is drawn. The situation occurs frequently in Data Processing and might be termed 'arboricide' : the murder of trees.

Consider as an example the specification of the file F given below. The file can have four different types of record, and their permitted interrelationship is given:

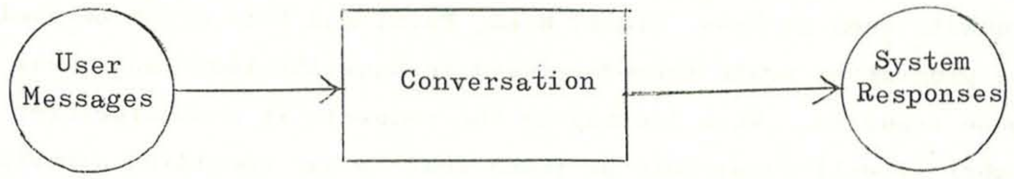
File F has records U, V, X, Y.

- U may follow X, Y, V.
- V may follow X, Y, V, U.
- X may be first or follow X, Y, V.
- Y may be first or follow X, Y, V.
- V is always the last record.

However, the structure imposed is not obvious until the corresponding tree is drawn:

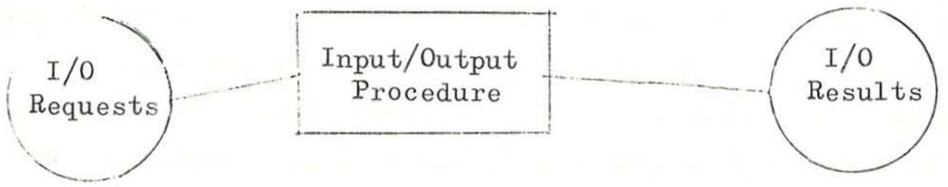


Here, it can be seen that the file F is defined to have the sequence 'A followed by B followed by C', where A, B, C and D are not explicitly mentioned in the specification above. 'A' is either an X or Y, C is a U followed by a V, and B is an iteration of D's, D being either an X, Y or V. The use of Program Inversion is helpful in avoiding this sort of arboricide. Consider the example in the figure below of a conversational system where the user types messages, the system responds, and the user enters another message depending on this reply.



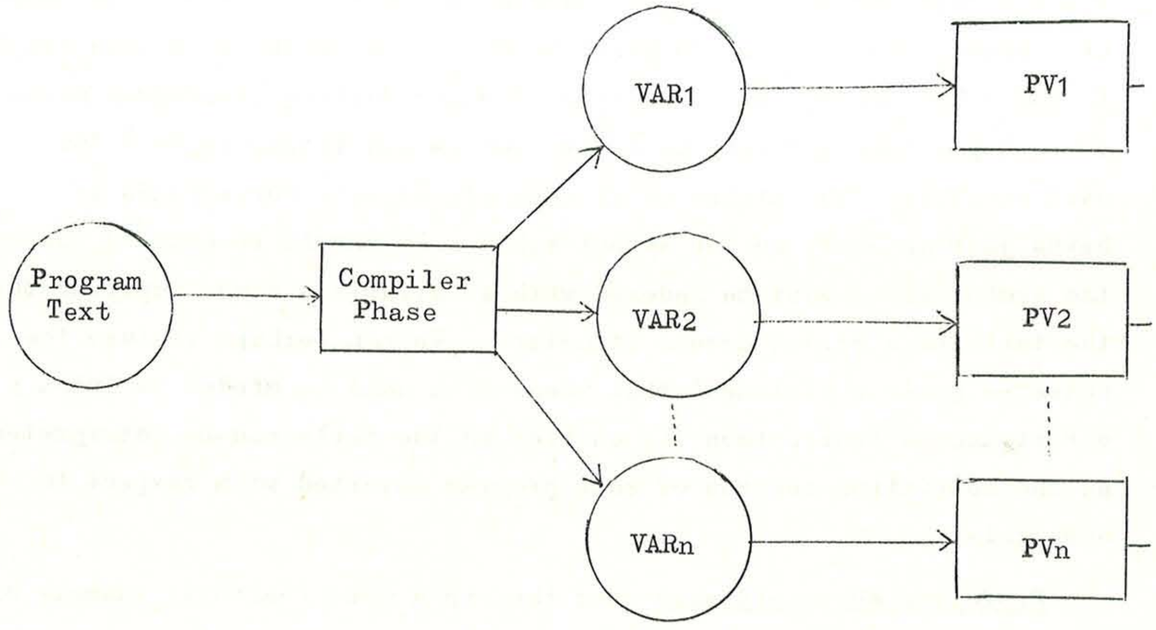
This could be implemented by writing a 'transaction processing module' where there might be one module for each type of input message, but it is preferable to regard the conversation program as a 'simple program' which can be implemented using Program Inversion. If it is inverted with respect to the file of user messages to produce a procedure whose specification is 'process the next message in the next conversation' then to extend the system to allow more than one conversation to be carried on only one copy of this program is needed, together with a state-vector (or activation record) for each user. The concern here is with where the design should start: if the programmer does not interpret the user message 'file' as a single data object which has a structure whose leaves are individual user messages, then he will be committing arboricide and will never be able to impose the right grammar on the user messages and system responses.

A similar example shown below is to provide an I/O procedure (known to IBM as an access method) for reading from and writing to files.



It also is a 'simple program'. The correct way to interpret the problem therefore is to invert the I/O procedure with respect to the file of I/O requests, producing an access method whose state-vector is a record of the history of the requests previously made. Incidentally, in using more complex transput at the assembler level, requests such as Open, Close, Read, Point and Note might be used; the programmer would therefore need to know the legal sequences of these requests. When looking up the requests in a manufacturer's manual it will invariably be found that he has committed arboricide by placing each procedure definition on a separate page so that the interrelationship between the various requests is very obscure.

A somewhat different problem arises in the next example: a compiler is needed for a language that distinguishes between numeric and non-numeric variables. The operations allowed on a variable depend on its type. For example, if it is declared to be numeric the operations of Add, Subtract, and Move might be allowed, whereas a non-numeric variable could only use Move. It is assumed that all structure and boundary clashes involved in disentangling the syntactic units at various levels have been dealt with, and the compiler has now just to deal with the variables. The situation is then a classic interleaving clash: it is easy to follow one particular variable through its declaration and use, but unfortunately the variables are interleaved throughout the program text. The problem is dealt with by reading the program text into some phase of the compiler, and splitting the input into several output files, VAR1, VAR2, ..., one per variable as in the figure below.



The appearance of any particular VARi file will be as in Figure 14:

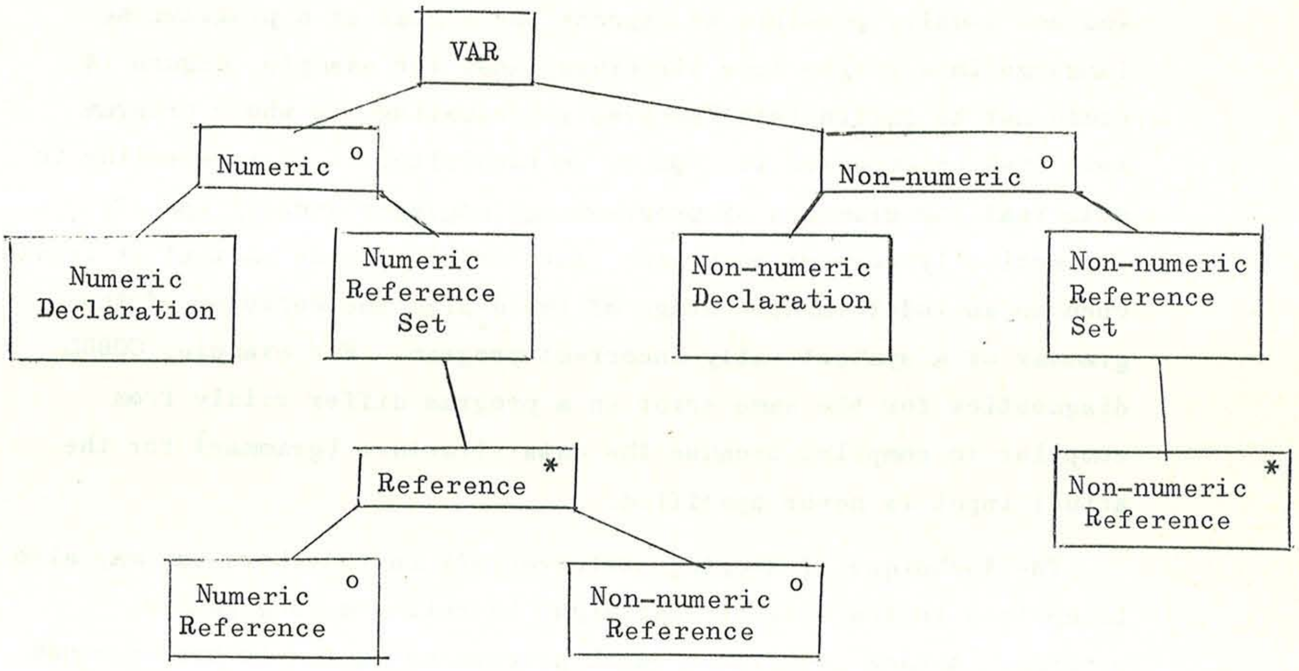
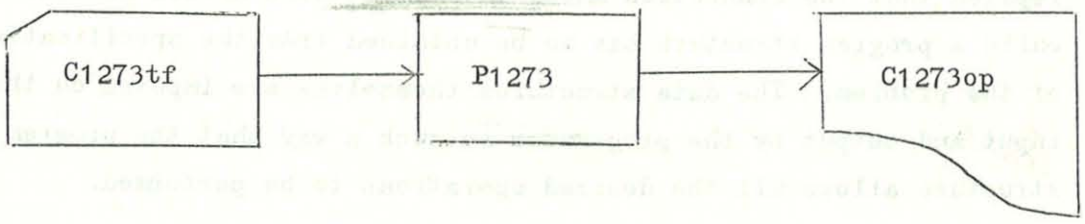


Figure 14

In an actual implementation it would be desirable to reduce the number of files needed, and so the programs PVi are each inverted with respect to their VARi file. Only one copy of the resulting PV program needs to be kept, together with an activation record (state-vector) for each variable. The state-vector then effectively corresponds to being just an entry in the symbol table. It should be noticed that the symbol table must be indexed with a variable's name, implying that the table is a direct access structure. We can perhaps venture the converse general statement that whenever a name is needed to index a direct access table, then the entries in the table can be interpreted as the activation records of some program inverted with respect to some file.

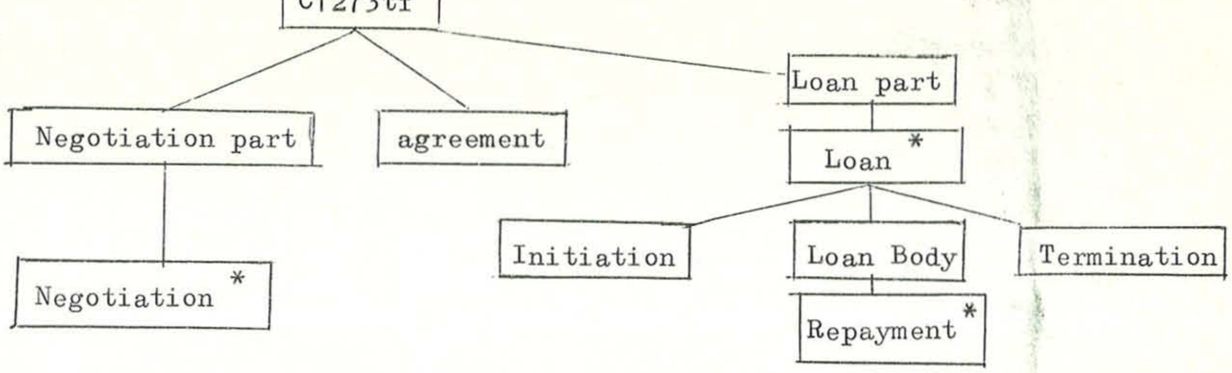
Professor Whitfield suggested that in a more realistic example of a compiler it would not be easy to find an appropriate data structure for the input or output, and to say that this would be solved by splitting the system into simple programs begged the issue since the problem of decomposition still remained. Mr. Jackson replied that it was not usually possible to express the syntax of a programming language in a single tree structure, and, for example, Figure 14 could not be fitted into the tree representing the whole program text that encompasses it. Quite incidentally, it is interesting to note that the grammars of programming languages usually specify syntactically correct programs. This is disastrous in that it leaves open to an indeterminate stage of the design the defining of the grammar of a syntactically incorrect program. For example, COBOL diagnostics for the same error in a program differ wildly from compiler to compiler because the data structure (grammar) for the actual input is never specified.

The technique of keeping state-vectors in the above way may also be applied in the case of what might be called a long-running program. A bank requires a batch processing system to keep account of its loans to its customers. The history of any one customer starts with the negotiations over general terms, ending in some sort of agreement after which the customer may have any number of loans, though only one at a time, each loan having an initiation followed by a period of repayments, and a termination. The important aspect of this problem is the timescale involved. Repayments can span several decades, and customers can exist for even centuries, yet what is required is a batch processing system with daily updating. The correct way to interpret this system is to consider first of all the case of just one customer, as represented below.



A program P1273 has been written for customer 1273 and runs on a dedicated machine for the lifetime of the customer-bank relation, say several decades. The program reads one record from the customer's file of transactions C1273tf once every ten weeks, and produces output such as acknowledgements on the file C1273op.

This simply way of looking at the problem makes clear that the structure of the file C1273tf is as shown below: there is a negotiation part consisting of several negotiations, followed by agreement, and a loan part of many loans, each having an initiation and termination with a body of many repayments.



Having written the program P1273 it can be applied to any other customer, and a real implementation requires only one copy of it to be kept together with a state-vector for each customer which would be preserved on some file, hence solving the problem.

Professor Whitfield wondered whether the Program Inversion method was effective: even given the input and output structures, was there an algorithm to determine if a structure clash existed? Mr. Jackson replied that the connection among data structures needed in order to build a program structure has to be obtained from the specification of the problem. The data structures themselves are imposed on the input and output by the programmer in such a way that the program structure allows all the desired operations to be performed.

PART III

In order to understand the environment into which we place both the ideas proposed earlier and those now to be introduced it is important to appreciate that in large Data Processing installations it is usual to form a group, commonly known as something like 'Systems Support' into which most of the more talented programmers are drawn: here they concern themselves with the new releases of software, not applications programming or analysis, which they often consider as not worthy of their attentions.

To summarise what has been said so far, the objective is to teach a design procedure, not an 'outlook', 'approach', or 'style'. At its simplest and most naïve level when applied to a small problem and small program there are three stages: first, design the data structures; from these form the program structure, then form a list of operations required and fit these into the program structure.

This is sufficient if the problem can be solved by a single 'simple program', but if not, then it becomes necessary to recognise the existence of a structure clash and decompose the problem as indicated. There appear to be three catagories of structure clash: boundary clashes, interleaving (which is extremely common), and an ordering clash, in which two structures have the same elements but not in the same order. It may be noted that should the elements be partially ordered it is possible to blur the distinction between interleaving and ordering clashes. Certain sorting algorithms, for example the pocket sort, can be regarded in this way.

There is a third level of design procedure which, as it is not yet sufficiently formalised, might be better classed as an approach, that derives from a less formal view of the second level. The second level is simply inadequate when required to aid the design of a very large system when the number of 'simple programs', structure classes and files may be several orders of magnitude above those for which the second level is intended, and such a scale is typical of a Data Processing environment. When the problem of the blocked file was considered the solution corresponded to something 'in the real world'. It was possible to imagine someone in a room stacking up cards group by group, and in another room a completely different person independently putting these groups into blocks. These two people are operating independently: the only thing passing between them is the complete file of cards. This independence is mirrored by the independence of the two 'simple programs'. To generalise this, a model of 'the real world' is constructed in which the individual independent entities are recognised. Corresponding to these should be simple programs. Where it is possible to form meaningful and relevant sets of entities in the real world, it is also possible to form a set of programs, that is, a single program text with a number of activation records.

Relating these remarks to those taught on the courses, it should be noted a number of points are stressed. Firstly, the programmer should think about the static, rather than the dynamic, aspects of the problem and, similarly, about the structures involved rather than the logic flow. In concentrating on structure in this way the aim is to avoid unwanted interactions. In 'the real world' the entities are connected only by some flow of transactions, hence the corresponding programs must be connected only in that way. Experience indicates a considerable proportion of the errors in systems arise because of unwanted interactions. For example, if it was attempted to solve the blocked file problem in the most obvious manner by amking the deblocking operations part of the same program as the disection into groups and formation of the total line, it could arise that the result would be a program which worked correctly except in the case when a block contains only one record, and that record is a header for a group containing no detail records.

If, however, the program is decomposed according to the methodology advocated, there is nothing passed between the two programs such that this could conceivably occur. The error is one of interaction between blocks and groups, and one program knows nothing about blocks and the other nothing about groups.

But, as pointed out earlier, backtracking problems can arise, and here this kind of error can be introduced because in essence one is forced to introduce 'GOTO' statements (albeit concealed). The problem is eased, fortunately, by having the correct non-backtracking text as a guide, this being formed by assuming backtracking can never occur.

When it was stated that the students were urged always to think about data rather than function, Dr. Pyle felt this needed clarifying: was function being used in the sense of specification? Mr. Jackson replied that the sense was that illustrated by maintaining the decomposition from the original problem into the simple was a functional decomposition. Dr. Pyle wished to contrast this, as typified by 'Put X into Y' with the job specification explaining the rationale. Mr. Jackson acknowledged the distinction but considered both unsatisfactory in terms of the design process.

Two other essential points remained to be stressed to the students. First, trees had to be considered, rather than leaves, in order to avoid arboricide, and second, emphasise the correct approach to optimisation.

The optimisation process was a fundamentally human process, perhaps with mechanical assistance, at the current state of the art. As an illustration, consider the following program fragment, assumed to be structurally perfect.

```
A sequence
    AA select condition-1
         p:=q;
         r:=s;
         t:=u;
    AA or condition-2
         v:=w;
         x:=y;
    AA or condition-3
         r:=s;
         t:=u;
    AA end
    AB iterate until condition-4
         m:=n;
         read file;
         v:=w;
         x:=y;
    AB end
A end
```

Clearly, some of the statements are repeated. By restructuring it is possible to remove some of the repetition:

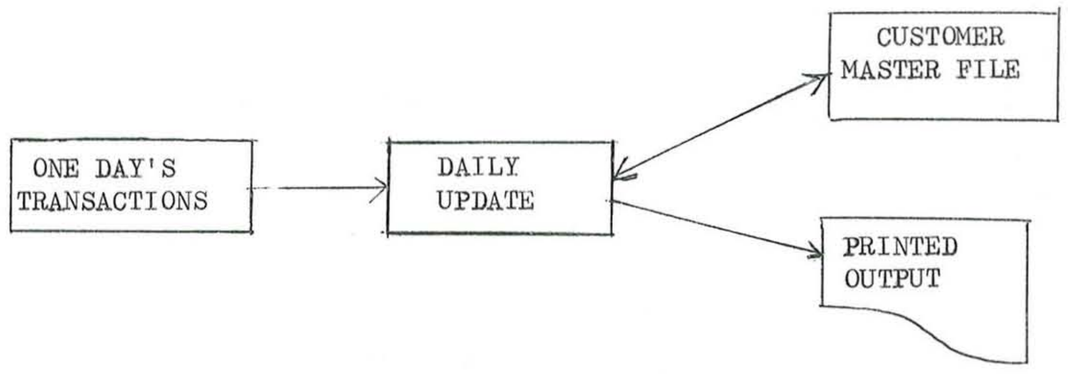
```
A sequence
    AA sequence
         AAA select condition-1
                  p:=q;
         AAA end
         AAB select condition-2
                  v:=w;
                  x:=y;
         AAB or true
                  r:=s;
                  t:=u;
         AAB end
    AA end
    AB iterate until condition-4
         m:=n;
         read file;
         v:=w;
         x:=y;
    AB end
A end
```

But this is confusing optimisation and program structure: moreover it does not appear obvious how we can remove the remaining duplication and stay within the structure programming techniques. If one is permitted the use of 'GOTO' statements it becomes trivial to remove both duplications:

```
A sequence
     AA select condition-1
          p:=q;
          goto OPT1;
     AA or condition-2
          goto OPT2;
     AA or condition-3
          OPT1: r:=s;
                t:=u;
     AA end
     AB iterate until condition-4
          m:=n;
          read in file
          OPT2: v:=w;
                x:=y
     AB end
A end
```

When should this optimisation be done? If the software to do it is not available then either it should not be done at all or it should be done in such a way that no-one is in any doubt that optimisation has been done. It is far better that software should handle the optimisation, but there do not appear to be compilers commonly available to do so, partly because it is assumed to be too difficult, and partly because of the confusion between design and optimisation.

The subject of the lecture was then widened to cover some of the ideals, rather than the practice of Data Processing.



The above represents a typical fragment of a Data Processing system, and it is conventional to present the problem in this way. This is wrong approach: it should be viewed on a customer basis paying attention to the program which simulates a single customer. An exact analogy to the conventional approach would be a compiler which takes cards from different locations in 20ms portions, compiles those until the time slice is exhausted and repeats the process on a possibly changing population of programs. A far better approach to the data processing problem (of which the daily update program is a fragment) is illustrated in Figure 15. The system is seen in a standardised form, free from implementation and efficiency considerations.

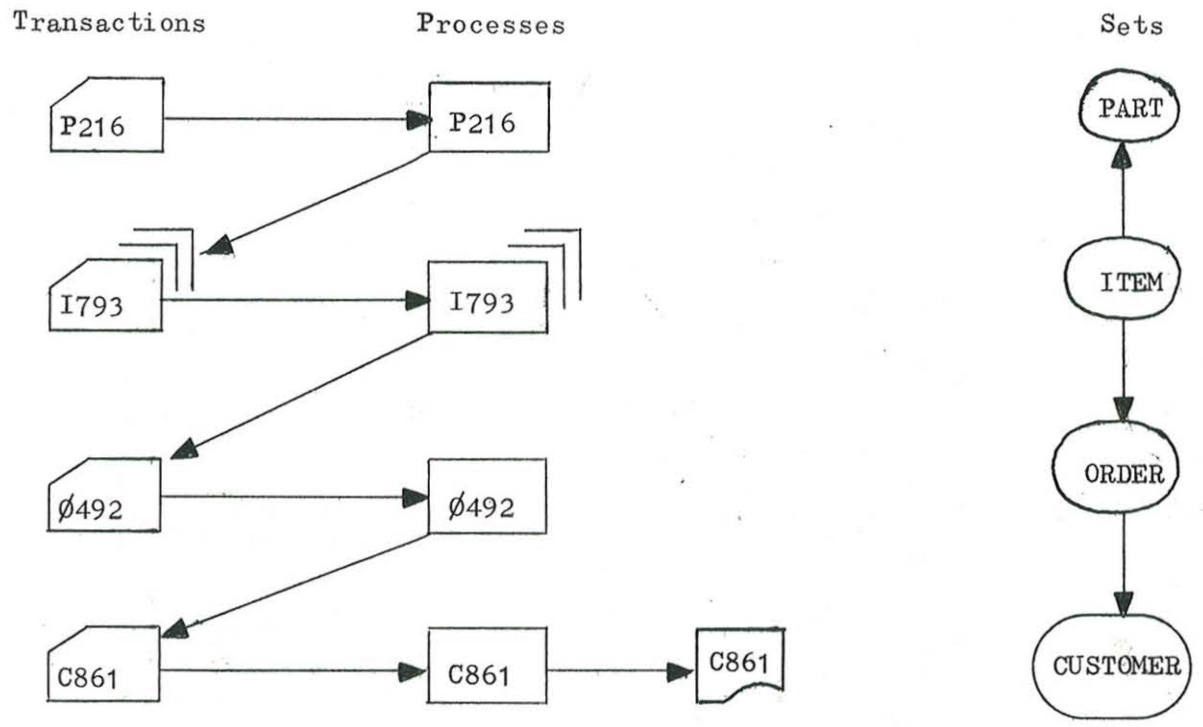


Figure 15

Customers place orders containing order items. Each order item refers to a part. The arrows are used to indicate 'many-to-one'. For example, one item can mention only one part. A price increase in part 216 will be a transaction, generating a transaction on item 793, (among others) which in its turn generates a transaction on order 492, hence to customer 861 who will receive a note stating that, due to a price increase in part 216 ordered in item 793 of order 492 the total cost has increased, and so on. Each of the rectangles in the diagram above represents a process (simple program) modelling the lifetime of the corresponding entity in the real world. The implementation question then arises: how should the process be scheduled? One common answer is provided by the 'transaction-oriented' design of Figure 16.

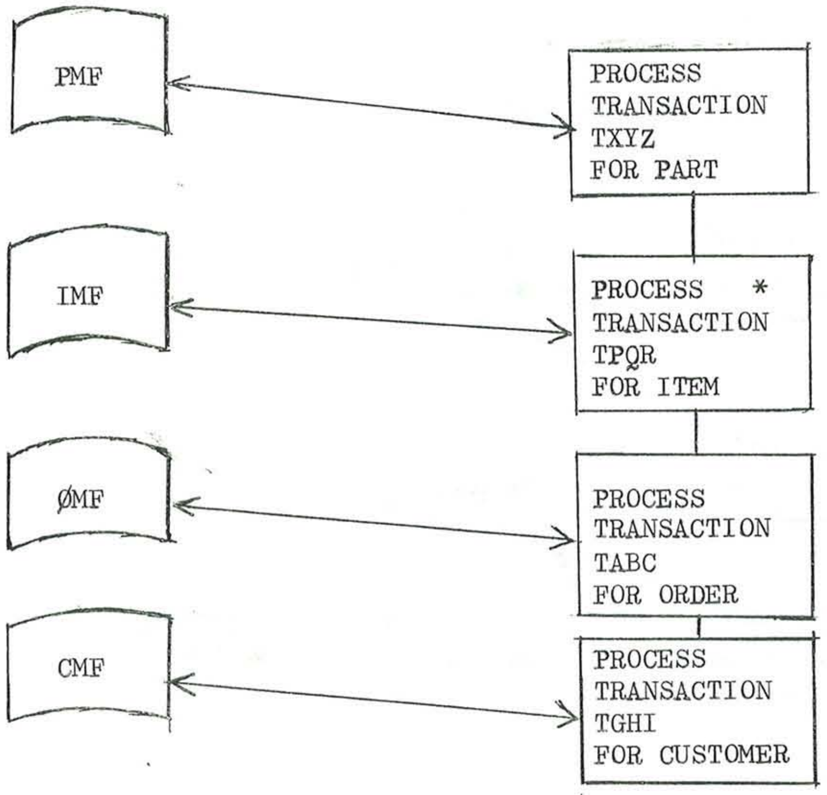

PMF
IMF
ØMF
CMF
PROCESS TRANSACTION TXYZ FOR PART
PROCESS * TRANSACTION TPQR FOR ITEM
PROCESS TRANSACTION TABC FOR ORDER
PROCESS TRANSACTION TGHI FOR CUSTOMER

Figure 16

Notice that here only a part of each simple process has been activated - the part relevant to a particular transaction: the possibility of errors becomes significant.

Another solution commonly adopted is to store up transactions for a period then process them as a group. This gives rise to a classical batch processing system as illustrated in Figure 17.

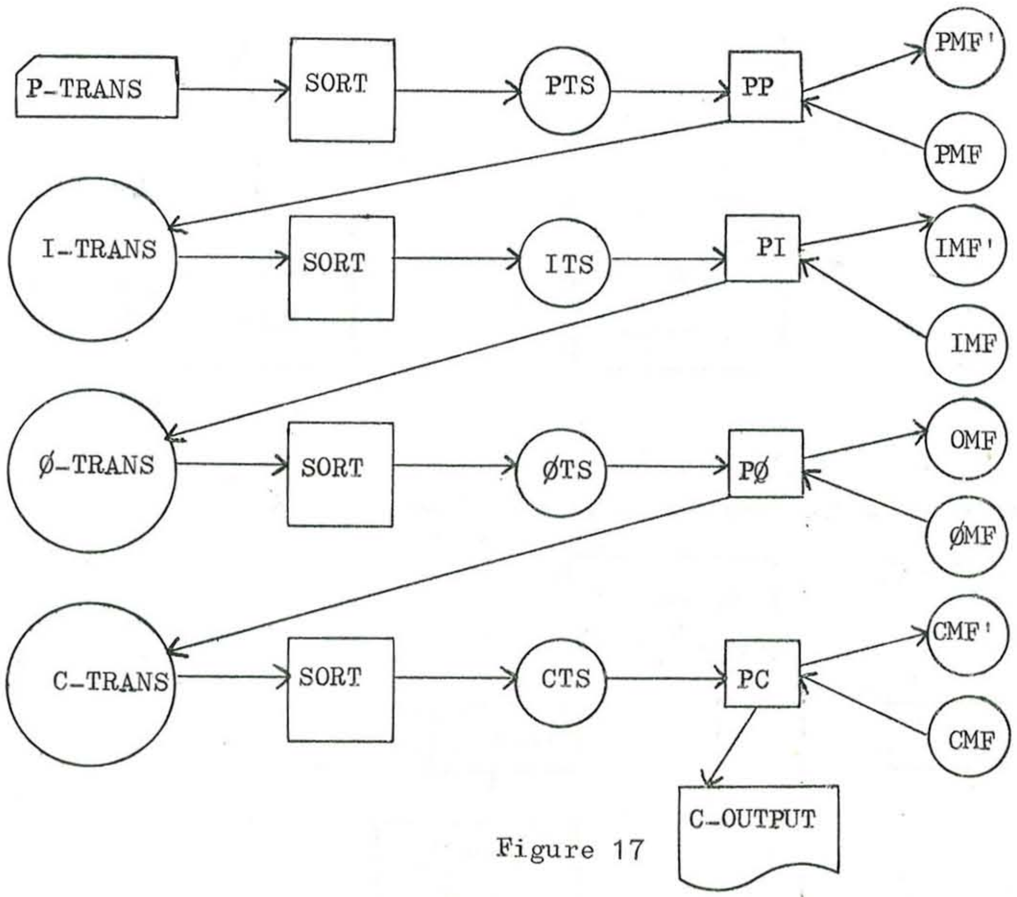


Figure 17

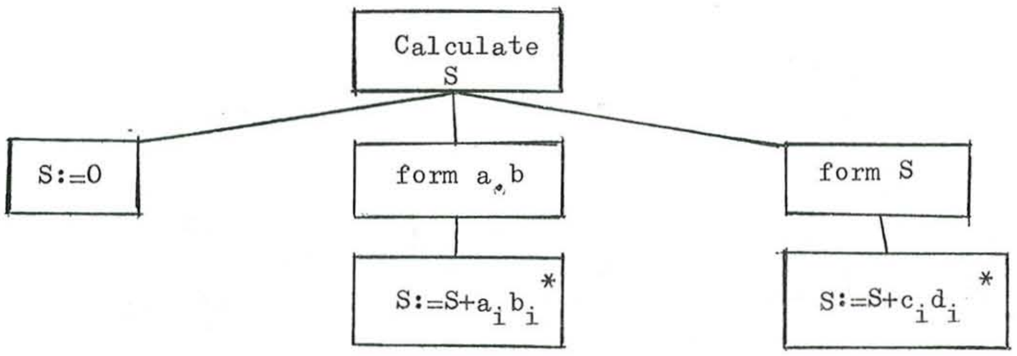
We can see both of these solutions as implementations (or, indeed, optimisations) of the standardised form. It would be very desirable to be able to produce the solutions by transforming the text of standardised form into the particular program texts required.

Once recent piece of work in the field of optimisation which may prove relevant is due to Burstall & Darlington.

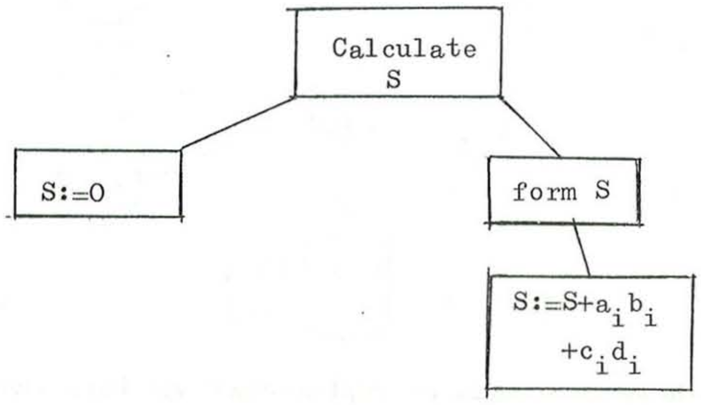
It is simply exemplified by the following. Suppose it is desired to calculate

$$S := (a \cdot b) + (c \cdot d)$$

where a, b, c and d are all vectors with the same number of elements. The calculation of a.b and c.d are independent processes. So the natural calculation is



But it is clearly possible to calculate it by



This means that it is possible to find two totally different control structures that have the same end effect. The work is concerned with how to find these alternative structures; it should have a major effect on programming, particularly on those systems which can be viewed as a transformations of networks of simple programs.