Proceedings of the 1st Conference of the European Cooperation in Informatics, Amsterdam August 9-12, 1976, pages 236-262; Lecture Notes in Computer Science 44, K Samelson ed, Springer, 1976.

CONSTRUCTIVE METHODS OF PROGRAM DESIGN

M. A. Jackson
Michael Jackson Systems Limited
101 Hamilton Terrace, London NW8

Abstract

Correct programs cannot be obtained by attempts to test or to prove incorrect programs: the correctness of a program should be assured by the design procedure used to build it.

A suggestion for such a design procedure is presented and discussed. The procedure has been developed for use in data processing, and can be effectively taught to most practising programmers. It is based on correspondence between data and program structures, leading to a decomposition of the program into distinct processes. The model of a process is very simple, permitting use of simple techniques of communication, activation and suspension. Some wider implications and future possibilities are also mentioned.

1. Introduction

In this paper I would like to present and discuss what I believe to be a more constructive method of program design. The phrase itself is important; I am sure that no-one here will object if I use a LIFO discipline in briefly elucidating its intended meaning.

'Design' is primarily concerned with structure; the designer must say what parts there are to be and how they are to be arranged. The crucial importance of modular programming and structured programming (even in their narrowest and crudest manifestations) is that they provide some definition of what parts are permissible: a module is a separately compiled, parameterised subroutine; a structure component is a sequence, an iteration or a selection. With such definitions, inadequate though they may be, we can at least begin to think about design: what modules should make up that program, and how should they be arranged? should this program be an iteration of selections or a sequence of iterations? Without such definitions, design is meaningless. At the top level of a problem there are P^N possible designs, where P is the number of distinct types of permissible part and N is the number of parts needed to make up the whole. So, to preserve our sanity, both P and N must be small: modular programming, using tree or hierarchical structures, offers small values of N; structured programming offers, additionally, small values of P.

'Program' or, rather, 'programming' I would use in a narrow sense. Modelling the problem is 'analysis'; 'programming' is putting the model on a computer. Thus, for example, if we are asked to find a prime number in the range 1050 to 1060, we need a number theorist for the analysis; if we are asked to program discounted cash flow, the analysis calls for a financial expert. One of the major ills in data processing stems from uncertainty about this distinction. In mathematical circles the distinction is often ignored altogether, to the detriment, I believe, of our understanding of programming. Programming is about computer programs, not about number theory, or financial planning, or production control.

'Method' is defined in the Shorter OED as a 'procedure for attaining an object'. The crucial word here is 'procedure'. The ultimate method, and the ultimate is doubtless unattainable, is a procedure embodying a precise and correct algorithm. To follow the method we need only execute the algorithm faithfully, and we will be led infallibly to the desired result. To the extent that a putative method falls short of this ideal it is less of a method.

To be 'constructive', a method must itself be decomposed into distinct steps, and correct execution of each step must assure correct execution of the whole method and thus the correctness of its product. The key requirement here is that the correctness of the execution of -a step should be largely verifiable without reference to steps not yet executed by the designer. This is the central difficulty in stepwise refinement: we can judge the correctness of a refinement step only by reference to what is yet to come, and hence only by exercising a degree of foresight to which few people can lay claim.

Finally, we must recognise that design methods today are intended for use by human beings: in spite of what was said above about constructive methods, we need, now and for some time to come, a substantial

ingredient of intuition and subjectivity. so what is presented below does not claim to be fully constructive — merely to be 'more constructive'. The reader must supply the other half of the comparison for himself, measuring the claim against the yardstick of his own favoured methods.

2. Basis of the Method

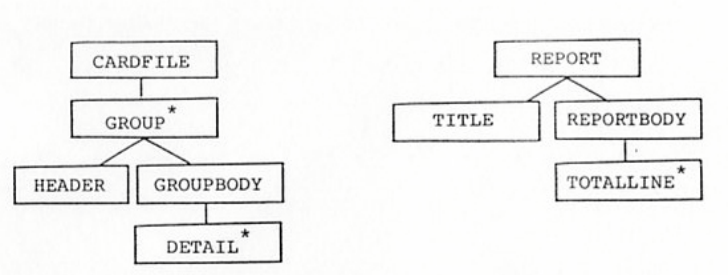
The basis of the method is described, in some detail, in (1). It is appropriate here only to illustrate it by a family of simple example problems.

Example 1

A cardfile of punched cards is sorted into ascending sequence of values of a key which appears in each card. Within this sequence, the first card for each group of cards with a common key value is a header card, while the others are detail cards. Each detail card carries an integer amount. It is required to produce a report showing the totals of amount for all keys.

Solution 1

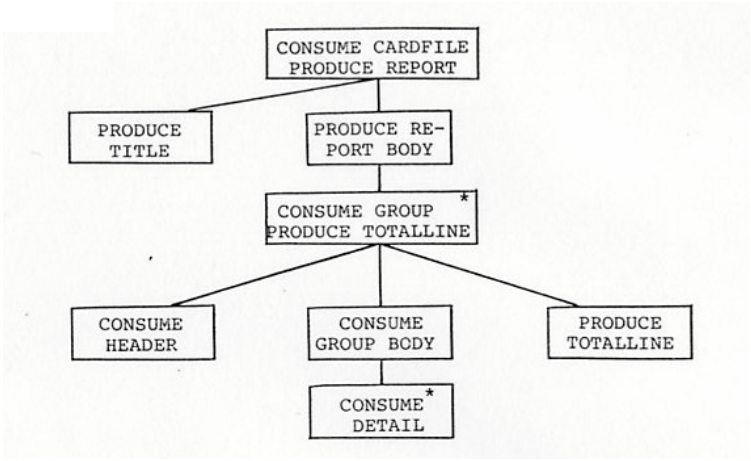
The first step in applying the method is to describe the structure of the data. We use a graphic notation to represent the structures as trees:–



The above representations are equivalent to the following (in BNF with iteration instead of recursion):

```
<cardfile> ::= <group>*
<group> ::= <header><groupbody>
<groupbody> ::= <detail>*
<report> ::= <title><reportbody>
<reportbody> ::= <totalline>*
```

The second step is to compose these data structures into a program structure:–



This structure has the following properties:

- It is related quite formally to each of the data structures. We may recover any one data structure from the program structure by first marking the leaves corresponding to leaves of the data structure, and then marking all nodes lying in a path from a marked node to the root.
- The correspondences (cardfile ≈ report) and (group ≈ totalline) are determined by the problem statement. One report is derivable from one cardfile; one totalline is derivable from one group, and the totallines are in the same order as the groups.
- The structure is vacuous, in the sense that it contains no executable statements: it is a program which does nothing; it is a tree without real leaves.

The third step in applying the method is to list the executable operations required and to allocate each to its right place in the program structure. The operations are elementary executable statements of the programming language, possibly after enhancement of the language by a bout of bottom-up design; they are enumerated, essentially, by working back from output to input along the obvious data-flow paths. Assuming a reasonably conventional machine and a line printer (rather than a character printer), we may obtain the list:

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

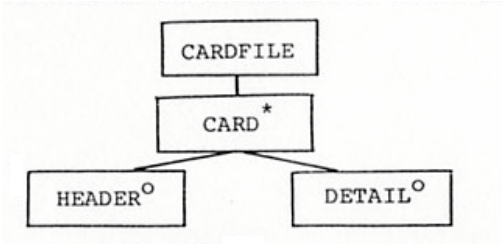
Note that every operation, or almost every operation, must have operands which are data objects. Allocation to a program structure is therefore a trivial task if the program structure is correctly based on the data structures. This triviality is a vital criterion of the success of the first two steps. The resulting program, in an obvious notation, is:

```
CARDFILE-REPORT sequence
  open cardfile; read cardfile; write title;
  REPORT-BODY iteration until cardfile.eof
    total := 0; groupkey := header.key; read cardfile;
      GROUP-BODY iteration until cardfile.eof or detail.key /= groupkey
        total := total + detail.amount; read cardfile;
      GROUP-BODY end
      write totalline (groupkey, total);
  REPORT-BODY end
  close cardfile;
CARDFILE-REPORT end
```

Clearly, this program may be transcribed without difficulty into any procedural programming language.

Comment

The solution has proceeded in three steps: first, we defined the data structures; second, we formed them into a program structure; third, we listed and allocated the executable operations. At each step we have criteria for the correctness of the step itself and an implicit check on the correctness of the steps already taken. For example, if at the first step we had wrongly described the structure of cardfile as:–



(that is:

```
<cardfile> ::= <card>*
<card> ::= <header>|<detail> )
```

we should have been able to see at the first step that we had failed to represent everything we knew about the cardfile. If nonetheless we had persisted in error, we would have discovered it at the second step, when we would have been unable to form a program structure in the absence of a cardfile component corresponding to totalline in report.

The design has throughout concentrated on what we may think of as a static rather than a dynamic view of the problem: on maps, not on itineraries, on structures, not on logic flow. The logic flow of the finished program is a by-product of the data structures and the correct allocation of the 'read' operation. There is an obvious connection between what we have done and the design of a very simple syntax analysis phase in a compiler: the grammar of the input file determines the structure of the program which parses it. We may observe that the 'true' grammar of the cardfile is not context-free: within one

group, the header and detail cards must all carry the same key value. It is because the explicit grammar cannot show this that we are forced to introduce the variable groupkey to deal with this stipulation.

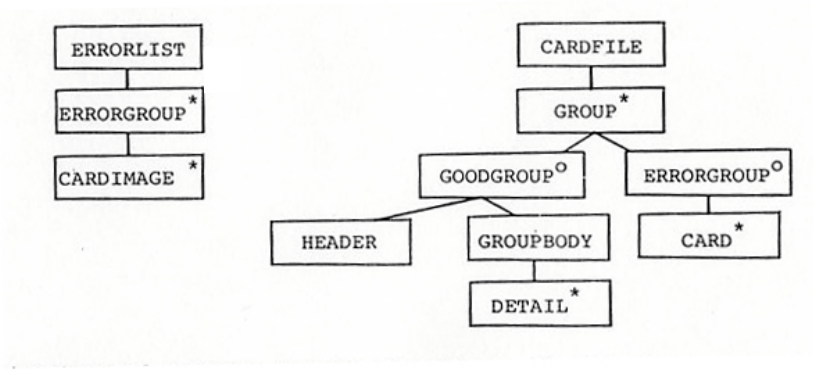
Note that there is no error-checking. If we wish to check for errors in the input we must elaborate the structure of the input file to accommodate those errors explicitly. By defining a structure for an input file we define the domain of the program: if we wish to extend the domain, we must extend the input file structure accordingly. In a practical data processing system, we would always define the structure of primary input (such as decks of cards, keyboard messages, etc) to encompass all physically possible files: it would be absurd to construct a program whose operation is unspecified (and therefore, in principle, unpredictable) in the event of a card deck being dropped or a wrong key depressed.

Example 2

The cardfile of example 1 is modified so that each card contains a cardtype indicator with possible values 'header', 'detail' and other. The program should take account of possible errors in the composition of a group: there may be no header card and/or there may be cards other than detail cards in the group body. Groups containing errors should be listed on an errorlist, but not totalled.

Solution 2

The structure of the report remains unchanged. The structure of the errorlist and of the new version of the cardfile are:–



The structure of cardfile demands attention. Firstly, it is ambiguous: anything which is a goodgroup is also an errorgroup. We are forced into this ambiguity because it would be intolerably difficult — and quite unnecessary — to spell out all of the ways in which a group may be in error. The ambiguity is simply resolved by the conventions we use: the parts of a selection are considered to be ordered, and the first applicable part encountered in a left-to-right scan is chosen. So a group can be parsed as an errorgroup only if it has already been rejected as a goodgroup. Secondly, a goodgroup cannot be recognised by a left-to-right parse of the input file with any predetermined degree of lookahead. If we choose to read ahead R records, we may yet encounter a group containing an error only in the R+1'th card.

Recognition problems of this kind occur in many guises. Their essence is that we are forced to a choice during program execution at a time when we lack the evidence on which the choice must be based. Note that the difficulty is not structural but is confined to achieving a workable flow of control. We will call such problems 'backtracking' problems, and tackle them in three stages:

(a) Ignore the recognition difficulty, imagining that a friendly demon will tell us infallibly which choice to make. In the present problem, he will tell us whether a group is a goodgroup or an errorgroup. Complete the design procedure in this blissful state of confidence, producing the full program text.
(b) Replace our belief in the demon's infallibility by a sceptical determination to verify each 'landmark' in the data which might prove him wrong. Whenever he is proved wrong we will execute a 'quit' statement which branches to the second part of the selection. These 'quit' statements are introduced into the program text created in stage (a).
(c) Modify the program text resulting from stage (b) to ensure that side-effects are repealed where necessary.

The result of stage (a), in accordance with the design procedure used for example 1, is:

```
CFILE-REPT-ERR sequence
  open cardfile; read cardfile; write title;
```

```
    REPORT-BODY iteration until cardfile.eof
      groupkey := card.key;
      GROUP-OUTG select goodgroup
        total := 0; read cardfile;
        GOOD-GROUP iteration until cardfile.eof or detail.key ≠ groupkey
          total := total + detail.amount; read cardfile;
        GOOD-GROUP end
        write totalline (qroupkey, total);
      GROUP-OUTG or errorgroup
        ERROR-GROUP iteration until cardfile.eof or card.key ≠ groupkey
          write errorline (card); read cardfile;
        ERROR-GROUP end
      GROUP-OUTG end
    REPORT-BODY end
    close cardfile;
  CFILE-REPT-ERR end
```

Note that we cannot completely transcribe this program into any programming language, because we cannot code an evaluable expression for the predicate goodgroup. However, we can readily verify the correctness of the program (assuming the infallibility of the demon). Indeed, if we are prepared to exert ourselves to punch an identifying character into the header card of each goodgroup — thus acting as our own demon — we can code and run the program as an informal demonstration of its acceptability.

We are now ready to proceed to stage (b,) in which we insert 'quit' statements into the first part of the selection GROUP-OUTG. Also, since quit statements are not present in a normal selection, we will replace the words 'select' and 'or' by 'posit' and 'admit' respectively, thus indicating the tentative nature of the initial choice. Clearly, the landmarks to be checked are the card-type indicators in the header and detail cards. We thus obtain the following program:

```
  CFILE-REPT-ERR sequence
    open cardfile; read cardfile; write title;
    REPORT-BODY iteration until cardfile.eof
      groupkey := card.key;
      GROUP-OUTG posit goodgroup
        total := 0;
        quit GROUP-OUTG if card.type ≠ header;
        read cardfile;
        GOOD-GROUP iteration until cardfile.eof or card.key ≠ groupkey
          quit GROUP-OUTG if card.type ≠ detail;
          total := total + detail.amount; read cardfile;
        GOOD-GROUP end
        write totalline (groupkey, total);
      GROUP-OUTG admit errorgroup
        ERROR-GROUP iteration until cardfile.eof or card.key ≠ groupkey;
         write errorline (card); read cardfile;
        ERROR-GROUP end
      GROUP-OUTG end
    REPORT-BODY end
    close cardfile;
  CFILE-KEPT-ERR end
```

The third stage, stage (c), deals with the side-effects of partial execution of the first part of the selection. In this trivial example, the only significant side-effect is the reading of cardfile. In general, it will be found that the only troublesome side-effects are the reading and writing of serial files; the best and easiest way to handle them is to equip ourselves with input and output procedures capable of 'noting' and 'restoring' the state of the file and its associated buffers. Given the availability of such procedures, stage (c) can be completed by inserting a 'note' statement immediately following the 'posit' statement and a 'restore' statement immediately following the 'admit'. Sometimes side-effects will demand a more *ad hoc* treatment: when 'note' and 'restore' are unavailable there is no alternative to such cumbersome expedients as explicitly storing each record on disk or in main storage.

Comment

By breaking our treatment of the backtracking difficulty into three distinct stages, we are able to isolate distinct aspects of the problem. In stage (a) we ignore the backtracking difficulty entirely, and

concentrate our efforts on obtaining a correct solution to the reduced problem. This solution is carried through the three main design steps, producing a completely specific program text: we are able to satisfy ourselves of the correctness of that text before going on to modify it in the second and third stages. In the second stage we deal only with the recognition difficulty: the difficulty is one of logic flow, and we handle it, appropriately, by modifying the logic flow of the program with quit statements. Each quit statement says, in effect, 'It is supposed (posited) that this is a goodgroup; but if, in fact, this card is not what it ought to be then this is not, after all, a goodgroup'. The required quit statements can be easily seen from the data structure definition, and their place is readily found in the program text because the program structure perfectly matches the data structure. The side-effects arise to be dealt with in stage (c) because of the quit statements inserted in stage (b): the quit statements are truly 'go to' statements, producing discontinuities in the context of the computation and hence side-effects. The side-effects are readily identified from the program text resulting from stage (b).

Note that it would be quite wrong to distort the data structures and the program structure in an attempt to avoid the dreaded four-letter word 'goto'. The data structures shown, and hence the program structure, are self-evidently the correct structures for the problem as stated: they must not be abandoned because of difficulties with the logic flow.

3. Simple Programs and Complex Programs

The design method, as described above, is severely constrained: it applies to a narrow class of serial file-processing programs. We may go further, and say that it defines such a class — the class of 'simple programs'. A 'simple program' has the following attributes:

- The program has a fixed initial state; nothing is remembered from one execution to the next.
- Program inputs and outputs are serial files, which we may conveniently suppose to be held on magnetic tapes. There may be more than one input and more than one output file.
- Associated with the program is an explicit definition of the structure of each input and output file. These structures are tree structures, defined in the grammar used above. This grammar permits recursion in addition to the features shown above; it is not very different from a grammar of regular expressions.
- The input data structures define the domain of the program, the output data structures its range. Nothing is introduced into the program text which is not associated with the defined data structures.
- The data structures are compatible, in the sense that they can be combined into a program structure in the manner shown above.
- The program structure thus derived from the data structures is sufficient for a workable program. Elementary operations of the program language (possibly supplemented by more powerful or suitable operations resulting from bottom-up design) are allocated to components of the program structure without introducing any further 'program logic'.

A simple program may be designed and constructed with the minimum of difficulty, provided that we adhere rigorously to the design principles adumbrated here and eschew any temptation to pursue efficiency at the cost of distorting the structure. In fact, we should usually discount the benefits of efficiency, reminding ourselves of the mass of error-ridden programs which attest to its dangers.

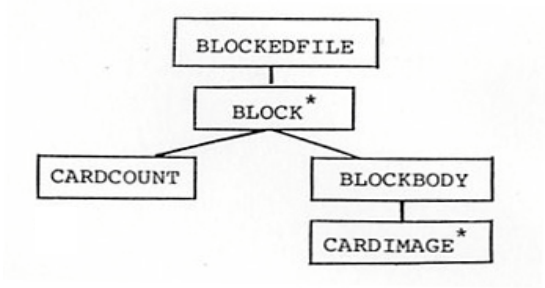
Evidently, not all programs are simple programs. Sometimes we are presented with the task of constructing a program which operates on direct-access rather than on serial files, or which processes a single record at each execution, starting from a varying internal state. As we shall see later, a simple program may be clothed in various disguises which give it a misleading appearance without affecting its underlying nature. More significantly, we may find that the design procedure suggested cannot be applied to the problem given because the data structures are not compatible: that is, we are unable at the second step of the design procedure to form the program structure from the data structures.

Example 3

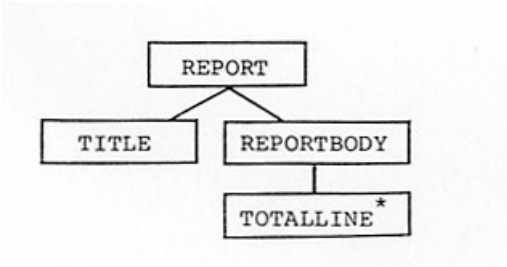
The input cardfile of example 1 is presented to the program in the form of a blocked file. Each block of this file contains a card count and a number of card images.

Solution 3

The structure of blockedfile is:

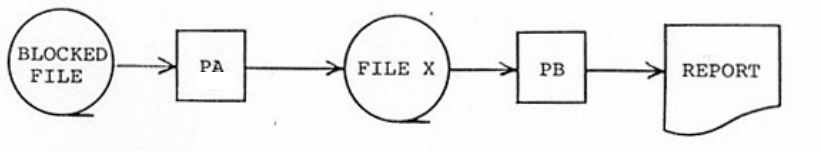


This structure does not, of course, show the arrangement of the cards in groups. It is impossible to show, in a single structure, both the arrangement in groups and the arrangement in blocks. But the structure of the report is still:



We cannot fit together the structures of report and blockedfile to form a program structure; nor would we be in better case if we were to ignore the arrangement in blocks. The essence of our difficulty is this: the program must contain operations to be executed once per block, and these must be allocated to a 'process block' component; it must also contain operations to be executed once per group, and these must be allocated to a 'process group' component; but it is impossible to form a single program structure containing both a 'process block' and a 'process group' component. We will call this difficulty a 'structure clash'.

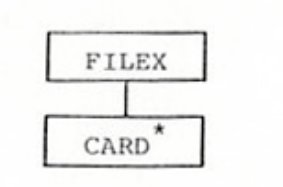
The solution to the structure clash in the present example is obvious: more so because of the order in which the examples have been taken and because everyone knows about blocking and deblocking. But the solution can be derived more formally from the data structures. The clash is of a type we will call 'boundary clash': the boundaries of the blocks are not synchronised with the boundaries of the groups. The standard solution for a structure clash is to abandon the attempt to form a single program structure and instead decompose the problem into two or more simple programs. For a boundary clash the required decomposition is always of the form:



The intermediate file, file X, must be composed of records each is a cardimage, because cardimage is the highest common structures blockedfile and cardfile. The program PB is the program produced as a solution to example 1; the program PA is:

```
PA sequence
  open blockedfile; open fileX; read blockedfile;
  PABODY iteration until blockedfile.eof
    cardpointer := 1;
    PBLOCK iteration until cardpointer > block.cardcount
      write cardimage (cardpointer); cardpointer := cardpointer + 1;
    PBLOCK end
    read blockedfile;
  PABODY end
  close fileX; close blockedfile;
PA end
```

The program PB sees file X as having the structure of cardfile in example 1, while program PA sees its structure as:



The decomposition into two simple programs achieves a perfect solution. Only the program PA is cognisant of the arrangement of cardimages in blocks; only the program PB of their arrangement in groups. The tape containing file X acts as a *cordon sanitaire* between the two, ensuring that no undesired interactions can occur: we need not concern ourselves at all with such questions as 'what if the header record of a group is the first cardimage in a block with only one cardimage?', or 'what if a group has no detail records and its header is the last cardimage in a block?'; in this respect our design is known to be correct.

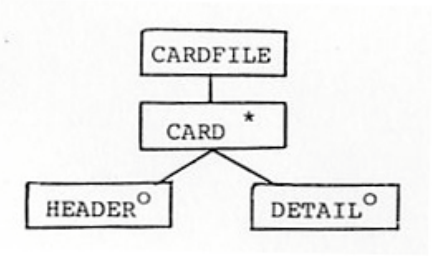
There is an obvious inefficiency in our solution. By introducing the intermediate magnetic tape file we have, to a first approximation, doubled the elapsed time for program execution and increased the program's demand for backing store devices.

Example 4

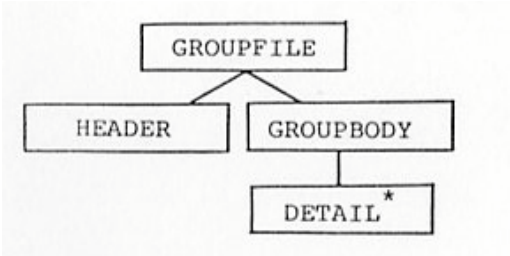
The input cardfile of example 1 is incompletely sorted. The cards are partially ordered so that the header card of each group precedes any detail cards of that group, but no other ordering is imposed. The report has no title, and the totals may be produced in any order.

Solution 4

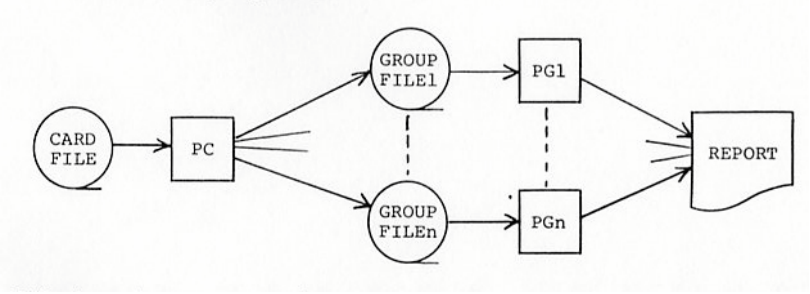
The best we can do for the structure of cardfile is:



which is clearly incompatible with the structure of the report, since there is no component of cardfile corresponding to totalline in the report. Once again we have a structure clash, but this time of a different type. The cardfile consists of a number of groupfiles, each one of which has the form:



The cardfile is an arbitrary interleaving of these groupfiles. To resolve the clash (an 'interleaving clash') we must resolve cardfile into its constituent groupfiles:



Allowing, for purposes of exposition, that a single report may be produced by the n programs PG1, ... PGn (each contributing one totalline), we have decomposed the problem into n+1 simple programs; of

these, n are identical programs processing the n distinct groupfiles groupfilel, ... groupfilen; while the other, PC, resolves cardfile into its constituents.

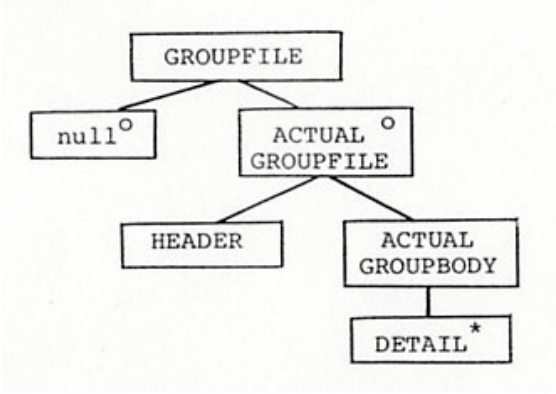
Two possible versions of PC are:

```
PC1 sequence
  open cardfile; read cardfile; open all possible groupfiles;
  PC1BODY iteration until cardfile.eof
    write record to groupfile (record.key); read cardfile;
  PC1BODY end
  close all possible groupfiles; close cardfile;
PC1 end
```

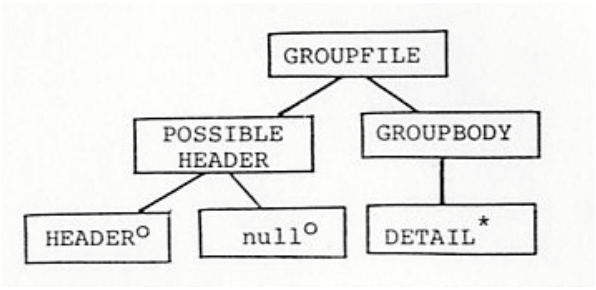
and

```
PC2 sequence
  open cardfile; read cardfile;
  PC2BODY iteration until cardfile.eof
    REC-INIT select new groupfile
      open groupfile (record.key);
   REC-INIT end
    write record to groupfile (record.key); read cardfile;
  PC2BODY end
  close all opened groupfiles; close cardfile;
PC2 end
```

Both PCI and PC2 present difficulties. In PCI we must provide a groupfile for every possible key value, whether or not cardfile contains records for that key. Also, the programs PG1, ... PGn must be elaborated to handle the null groupfile:



In PC2 we must provide a means of determining whether a groupfile already exists for a given key value. Note that it would be quite wrong to base the determination on the fact that a header must be the first record for a group: such a solution takes impermissible advantage of the structure of groupfile which, in principle, is unknown in the program PC; we would then have to make a drastic change to PC if, for example, the header card were made optional:



Further, in PC2 we must be able to run through all the actual key values in order to close all the groupfiles actually opened. This would still be necessary even if each group had a recognisable trailer record, for reasons similar to those given above concerning the header records.

Comment

The inefficiency of our solution to example 4 far outstrips the inefficiency of our solution to example 3. Indeed, our solution to example 4 is entirely impractical. Practical implementation of the designs will be considered below in the next section. For the moment, we may observe that the use of magnetic tapes for

communication between simple programs enforces a very healthy discipline. We are led to use a very simple protocol: every serial file must be opened and closed. The physical medium encourages a complete decoupling of the programs: it is easy to imagine one program being run today, the tapes held overnight in a library, and a subsequent program being run tomorrow; the whole of the communication is visible in the defined structure of the files. Finally, we are strengthened in our resolve to think in terms of static structures, avoiding the notoriously error-prone activity of thinking about dynamic flow and execution-time events.

Taking a more global view of the design procedure, we may say that the simple program is a satisfactory high level component. It is a larger object than a sequence, iteration or selection; it has a more precise definition than a module; it is subject to restrictions which reveal to us clearly when we are trying to make a single program out of what should be two or more.

4. Programs, Procedures and Processes

Although from the design point of view we regard magnetic tapes as the canonical medium of communication between simple programs, they will not usually provide a practical implementation.

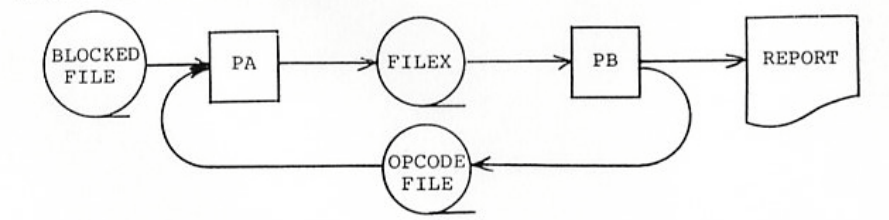
An obvious possibility for implementation in some environments is to replace each magnetic tape by a limited number of buffers in main storage, with a suitable regime for ensuring that the consumer program does not run ahead of the producer. Each simple program can then be treated as a distinct task or process, using whatever facilities are provided for the management of multiple concurrent tasks.

However, something more like coroutines seems more attractive (2). The standard procedure call mechanism offers a simple implementation of great flexibility and power. Consider the program PA, in our solution to example 3, which writes the intermediate file X. We can readily convert this program into a procedure PAX which has the characteristics of an input procedure for file X. That is, invocations of the procedure PAX will satisfactorily implement the operations 'open file X for reading', 'read file X' and 'close file X after reading'.

We will call this conversion of PA into PAX 'inversion of PA with respect to file X'. (Note that the situation in solution 3 is symmetrical: we could equally well decide to invert PB with respect to file X, obtaining an output procedure for file X.) The mechanics of inversion are a mere matter of generating the appropriate object coding from the text of the simple program: there is no need for any modification to that text. PA and PAX are the same program, not two different programs. Most practising programmers seem to be unaware of this identity of PA and PAX, and even those who are familiar with coroutines often program as if they supposed that PA and PAX were distinct things. This is partly due to the baleful influence of the stack as a storage allocation device: we cannot jump out of an inner block of PAX, return to the invoking procedure, and subsequently resume where we left off when we are next invoked. So we must either modify our compiler or modify our coding style, adopting the use of labels and go to statements as a standard in place of the now conventional compound statement of structured programming. It is common to find PAX, or an analogous program, designed as a selection or case statement: the mistake is on all fours with that of the kindergarten child who has been led to believe that the question 'what is 5 multiplied by 3?' is quite different from the question 'what is 3 multiplied by 5?'. At a stroke the poor child has doubled the difficulty of learning the multiplication tables.

The procedure PAX is, of course, a variable state procedure. The value of its state is held in a 'state vector' (or activation record), of which a vital part is the text pointer; the values of special significance are those associated with the suspension of PAX for operations on file X — open, write and close. The state vector is an 'own variable' par excellence, and should be clearly seen as such.

The minimum interface needed between PB and PAX is two parameters: a record of file X, and an additional bit to indicate whether the record is or is not the eof marker. This minimum interface suffices for example 3: there is no need for PB to pass an operation code to PAX (open read or close). It is important to understand that this minimum interface will not suffice for the general case. It is sufficient for example 3 only because the operation code is implicit in the ordering of operations. From the Point of view of PAX, the first invocation must be 'open', and subsequent invocations must be 'read' until PAX has returned the eof marker to PB, after which the final invocation must be 'close'. This felicitous harmony is destroyed if, for example, PB is permitted to stop reading and close file X before reaching the eof marker. In such a case the interface must be elaborated with an operation code. Worse, the sequence of values of this operation code now constitutes a file in its own right; the solution becomes:



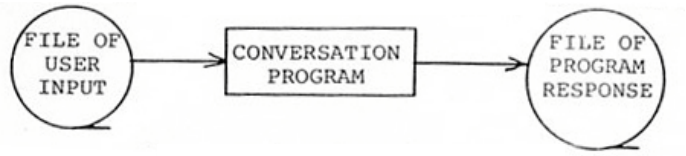
The design of PA is, potentially, considerably more complicated. The benefit we will obtain from treating this complication conscientiously is well worth the price: by making explicit the structure of the opcode file we define the problem exactly and simplify its solution. Failure to recognise the existence of the opcode file, or, just as culpable, failure to make its structure explicit, lies at the root of the errors and obscurities for which manufacturers' input-output software is deservedly infamous.

In solution 4 we created an intolerable multiplicity of files — groupfilel, ... groupfilen. We can rid ourselves of these by inverting the programs PG1, ... PGn with respect to their respective groupfiles: that is, we convert each of the programs PGi to an output procedure PGFi, which can be invoked by PC to execute operations on groupfilei. But we still have an intolerable multiplicity of output procedures, so a further step is required. The procedures are identical except for their names and the current values of their state vectors. So we separate out the pure procedure part — PGF — of which we need keep only one copy, and the named state vectors SVPGFI, ... SVPGFn. We must now provide a mechanism for storing and retrieving these state vectors and for associating the appropriate state vector with each invocation of PGF; many mechanisms are possible, from a fully-fledged direct-access file with serial read facilities to a simple arrangement of the state vectors in an array in main storage.

5. Design and Implementation

The model of a simple program and the decomposition of a problem into simple programs provides some unity of viewpoint. In particular, we may be able to see what is common to programs with widely different implementations. Some illustrations follow.

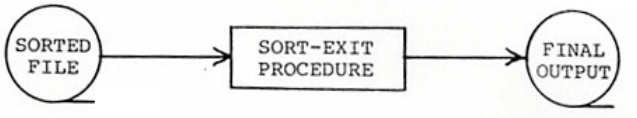
(d) A conversational program is a simple program of the form:



The user provides a serial input file of messages, ordered in time; the conversation program produces a serial file of responses. Inversion of the program with respect to the user input file gives an output procedure 'dispose of one message in a conversation'. The state vector of the inverted program must be preserved for the duration of the conversation: IBM's IMS provides the SPA (Scratchpad Area) for precisely this purpose. The conversation program must, of course, be designed and written as a single program: implementation restrictions may dictate segmentation of the object code.

(e) A 'sort-exit' allows the user of a generalised sorting program to introduce his own procedure at the point where each record is about to be written to the final output file. An interface is provided which permits 'insertion' and 'deletion' of records as well as 'updating'.

We should view the sort-exit procedure as a simple program:



To fit it in with the sorting program we must invert it with respect to both the sorted file and the final output. The interface must provide an implementation of the basic operations: open sortedfile for reading; read sortedfile (distinguishing the eof marker); close sortedfile after reading; open finaloutput for writing; write finaloutput record; close finaloutput file after writing (including writing the eof marker).

Such concepts as 'insertion' and 'deletion' of records are pointless: at best, they serve the cause of efficiency, traducing clarity; at worst, they create difficulty and confusion where none need exist.

(f) Our solution to example 1 can be seen as an optimisation of the solution to the more general example 4. By sorting the cardfile we ensure that the groups do not overlap in time: the state vectors of the inverted programs PGF1, ... PGFn can therefore share a single area in main storage. The state vector consists only of the variable total; the variable groupkey is the name of the currently active group and hence of the current state vector. Because the records of a group are contiguous, the end of a group is recognisable at cardfile.eof or at the start of another group. The individual groupfile may therefore be closed, and the totalline written, at the earliest possible moment.

We may, perhaps, generalise so far as to say that an identifier is stored by a program only in order to give a unique name to the state vector of some process.

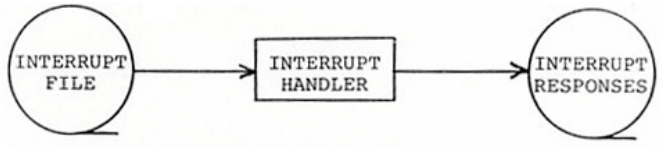
(g) A data processing system may be viewed as consisting of many simple programs, one for each independent entity in the real world model. By arranging the entities in sets we arrange the corresponding simple programs in equivalence classes. The 'master record' corresponding to an entity is the state vector of the simple program modelling that entity.

The serial files of the system are files of transactions ordered in time: some are primary transactions, communicating with the real world, some are secondary, passing between simple programs of the system. In general, the real world must be modelled as a network of entities or of entity sets; the data processing system is therefore a network of simple programs and transaction files.

Implementation of the system demands decisions in two major areas. First a scheduling algorithm must be decided; second, the representation and handling of state vectors. The extreme cases of the first are 'real-time' and 'serial batch'. In a pure 'real-time' system every primary transaction is dealt with as soon as it arrives, followed immediately by all of the secondary and consequent transactions, until the system as a whole becomes quiet. In a pure 'serial batch' system, each class (identifier set) of primary transactions is accumulated for a period (usually a day, week or month). Each simple program of that class is then activated (if there is a transaction present for it), giving rise to secondary transactions of various classes. These are then treated similarly, and so on until no more transactions remain to be processed.

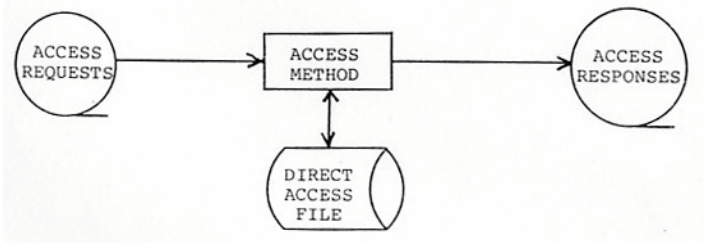
Choosing a good implementation for a data processing system is difficult, because the network is usually large and many possible choices present themselves. This difficulty is compounded by the long-term nature of the simple programs: a typical entity, and hence a typical program, has a lifetime measured in years or even decades. During such a lifetime the system will inevitably undergo change: in effect, the programs are being rewritten while they are in course of execution.

(h) An interrupt handler is a program which processes a serial file of interrupts, ordered in time:



Inversion of the interrupt handler with respect to the interrupt file gives the required procedure 'dispose of one interrupt'. In general, the interrupt file will be composed of interleaved files for individual processes, devices, etc. Implementation is further complicated by the special nature of the invocation mechanism, by the fact that the records of the interrupt file are distributed in main storage, special registers and other places, and by the essentially recursive structure of the main interrupt file (unless the interrupt handler is permitted to mask off secondary interrupts).

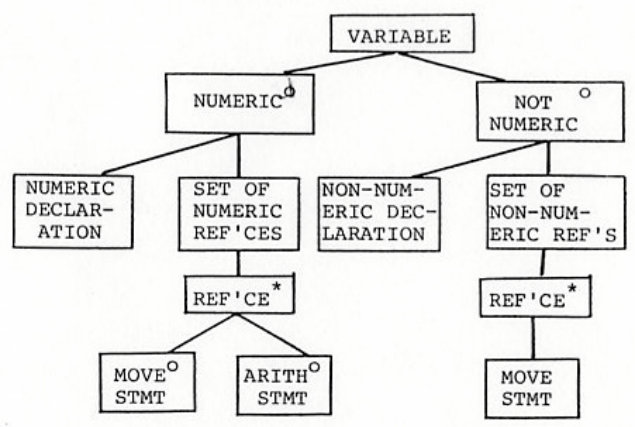
(i) An input-output procedure (what IBM literature calls an 'access method') is a simple program which processes an input file of access requests and produces an output file of access responses. An access request consists of an operation code and, sometimes, a data record; an access response consists of a result code and, sometimes, a data record. For example, a direct-access method has the form:



By inverting this simple program with respect to both the file of access requests and the file of access responses we obtain the desired procedure. This double inversion is always possible without difficulty, because each request must produce a response and that response must be calculable before the next request is presented.

The chief crime of access method designers is to conceal from their customers (and, doubtless, from themselves) the structure of the file of access requests. The user of the method is thus unable to determine what sequences of operations are permitted by the access method, and what their effect will be.

(j) Some aspects of a context-sensitive grammar may be regarded as interleaved context-free grammars. For example, in a grossly simplified version of the COBOL language we may wish to stipulate that any variable may appear as an operand of a MOVE statement, while only a variable declared as numeric may appear as an operand of an arithmetic (ADD, SUBTRACT, MULTIPLY or DIVIDE) statement. We may represent this stipulation as follows:



The syntax-checking part of the compiler consists, partly, of a simple program for each declared variable. The symbol table is the set of state vectors for these simple programs. The algorithm for activating and suspending these and other programs will determine the way in which one error interacts with another both for diagnosis and correction.

6. A Modest Proposal

It is one thing to propose a model to illuminate what has already been done, to clarify the sources of existing success or failure. It is quite another to show that the model is of practical value, and that it leads to the construction of acceptable programs. An excessive zeal in decomposition produces cumbersome interfaces and pointlessly redundant code. The “Shanley Principle” in civil engineering (3) requires that several functions be implemented in a single part; this is necessary for economy both in manufacturing and in operating the products of engineering design. It appears that a design approach which depends on decomposition runs counter to this principle: its main impetus is the separation of functions for implementation in distinct parts of the program.

But programs do not have the intractable nature of the physical objects which civil, mechanical or electrical engineers produce. They can be manipulated and transformed (for example, by compilers) in ways which preserve their vital qualities of correctness and modifiability while improving their efficiency both generally and in the specialised environment of a prticular machine. The extent to which a program can be manipulated and transformed is critically affected by two factors: the variety of forms it can take, and the semantic clarity of the text. Programs written using today’s conventional techniques score poorly on both factors. There is a distressingly large variety of forms, and intelligibility is compromised or even destroyed by the introduction of implementation-orientated features. The justification for these techniques is, of course, efficiency. But in pursuing efficiency in this way we become caught in a vicious circle: because our languages are rich the compilers cannot understand, and hence cannot optimise, our programs; so we need rich languages to allow us to obtain the efficiency which the compilers do not offer.

Decomposition into simple programs, as discussed above, seems to offer some hope of separating the considerations of correctness and modifiability from the considerations of efficiency. Ultimately, the objective is that the first should become largely trivial and the second largely automatic.

The first phase of design would produce the following documents:

- a definition of each serial file structure for each simple program (including files of operation codes!);
- the text of each simple program;
- a statement of the communication between simple programs, perhaps in the form of identities such as

```
output (Pi, fr) ≡ input (pj, fs).
```

It may then be possible to carry out some automatic checking of self-cons sistency in the design — for instance, to check that the inputs to a program are within its domain. We may observe, incidentally, that the 'inner' feature of Simula 67 (4) is a way of enforcing consistency of a file of operation codes between the consumer and producer processes in a very limited case. More ambitiously, it may be possible, if file-handling protocol is exactly observed, and read and write operations are allocated with a scrupulous regard to principle, to check the correctness of the simple programs in relation to the defined data structures.

In the second phase of design, the designer would specify, in greater or lesser detail:

- the synchronisation of the simple programs;
- the handling of state vectors;
- the dissection and recombining of programs and state vectors to reduce interface overheads.

Synchronisation is already loosely constrained by the statements of program communication made in the first phase: the consumer can never run ahead of the producer. Within this constraint the designer may choose to impose additional constraints at compile time and/or at execution time. The weakest local constraint is to provide unlimited dynamic buffering at execution time, the consumer being allowed to lag behind the producer by anything from a single record to the whole file, depending on resource allocation elsewhere in the system. The strongest local constraints are use of coroutines or program inversion (enforcing a single record lag) and use of a physical magnetic tape (enforcing a whole file lag).

Dissection and recombining of programs becomes possible with coroutines or program inversion; its purpose is to reduce interface overheads by moving code between the invoking and invoked programs, thus avoiding some of the time and space costs of procedure calls and also, under certain circumstances, avoiding replication of program structure and hence of coding for sequencing control. It depends on being able to associate code in one program with code in another through the medium of the communicating data structure.

A trivial illustration is provided by solution 3, in which we chose to invert PA with respect to file X, giving an input procedure PAX for the file of cardimages. We may decide that the procedure call overhead is intolerable, and that we wish to dissect PAX and combine it with FR. This is achieved by taking the invocations of PAX in PB (that is, the statements 'open fileX', 'read fileX' and 'close fileX') and replacing those invocations by the code which PAX would execute in response to them. For example, in response to 'open fileX', PAX would execute the code 'open blockedfile'; therefore the 'open fileX' statement in PB can be replaced by the statement 'open blockedfile'.

A more substantial illustration is provided by the common practice of designers of 'real-time' data processing systems. Suppose that a primary transaction for a product gives rise to a secondary transaction for each open order item for that product, and that each of those in turn gives rise to a transaction for the open order of which it is a part, which then gives rise to a transaction for the customer who placed the order. Instead of having separate simple programs for the product, order item, order and customer, the designer will usually specify a 'transaction processing module': this consists of coding from each of those simple programs, the coding being that required to handle the relevant primary or secondary transaction.

Some interesting program transformations of a possibly relevant kind are discussed in a paper by Burstall and Darlington (5). I cannot end this paper better than by quoting from them:

> "The overall aim of our investigation has been to help people to write correct programs which are easy to alter. To produce such programs it seems advisable to adopt a lucid, mathematical and abstract programming style. If one takes this really seriously, attempting to free one's mind from considerations of computational efficiency, there may be a heavy penalty in program running time; in practice it is often necessary to adopt a more intricate version of the program, sacrificing comprehensibility for speed. The question then arises as to how a lucid program can be transformed into a more intricate but efficient one in a systematic way, or indeed in a way which could be mechanised.

"... We are interested in starting with programs having an extremely simple structure and only later introducing the complications which we usually take for granted even in high level language programs. These complications arise by introducing useful interactions between what were originally separate parts of the program, benefiting by what might be called 'economies of interaction'."

References

(1) Principles of Program Design; M A Jackson; Academic Press 1975.

(2) Hierarchical Program Structures; O-J Dahl; in Structured Programming; Academic Press 1972.

(3) Structured Programming with go to Statements; Donald E Knuth; in ACM Computing Surveys Vol 6 No 4 December 1974.

(4) A Structural Approach to Protection; C A R Hoare; 1975.

(5) Some Transformations for Developing Recursive Programs; R M Burstall & John Darlington; in Proceedings of 1975 Conference on Reliable Software; Sigplan Notices Vol 10 No 6 June 1975.