THE CONSTRUCTION OF ALGORITHMS IN A COMMERCIAL PROGRAMMING ENVIRONMENT

Mr. M. A. Jackson

Hoskyns Systems Research Limited,
33, Chancery Lane,
London, W.C.2.

Abstract:

The commercial programming environment presents a class of problems whose complexity can easily become unmanageable. Creation of suitable programs is facilitated by careful structure. This paper presents structuring techniques which the author has found to be appropriate in this environment.

Rapporteurs:

Dr. J. J. Horning
Mr. J. G. Givens

1. Introduction

I would like to describe some of the problems facing people in a commercial programming environment and to describe some of our reactions to it. Unlike Mr. d'Agapeyeff, I do not feel that commercial programming is even remotely respectable. Some of the problems described will be trivial, and if their solutions are mundane that merely illustrates the sorry straits of commercial programming. There is a vast gap between the academic and commercial worlds. I hope you will bear with me if I describe problems and solutions in our own terms, rather than in the terms to which you are accustomed.

2. By 'commercial programming environment', is meant a certain type of business data processing (normally for commerce, industry or government) which includes such things as accounting (payroll, sales ledger, purchase ledger, invoicing, costing, etc.) and optimization (route planning, stock control, production planning, etc.). Before the advent of computers the accounting problems were solved by armies of clerks, so we may be sure that they are computationally trivial. The optimization problems have a tiny nucleus of sophisticated mathematics which is almost buried by other considerations.

What are the chief characteristics of these problems which determine the commercial programming environment?

1. There are large volumes of data (25,000,000 records is a possible file size for an insurance application). Therefore, efficiency is almost always a major concern.
2. Programs are not constructed in isolation, but as systems of programs, which are interdependent.
3. Most problems are in some sense human problems, stemming from unpredictable human behaviour and tastes.
4. Systems change all the time. New requirements keep them in a constant state of flux.
5. Systems must continually cope with invalid data, and purge the results of invalid data.
6. It is often necessary to process data retrospectively, correcting assumptions which have proved wrong.

These aspects are stressed because they are the ultimate source of most of the difficulties in the commercial programming environment.

3. Systems and programming

By a system, is meant a number of programs connected by a number of files. For example, the master employee file will be accessed by programs to calculate gross wages, process leavers and joiners, produce personnel statistics, record absences, etc. Note that different programs will access the same file differently. It would be expected that the number of leavers and joiners is small, so that program would probably use a direct access technique, while the volume of the payroll dictates that the file be accessed sequentially for calculating gross wages.

Note that files themselves have structure. A typical file might consist of a number of customer groups, each containing a number of order record groups, which in turn consist of a number of item records. These logically related records become physically related by placing them in the same file.

Normally, each program will use many files. Some of the files which might be needed by a program to calculate gross wages are: master employee file, time sheets, tables of piecework rates, etc. Furthermore, for reasons of efficiency, a single program may be required to do many disparate things. Axiomatically: 'We can't afford to pass the master file more than once'. Thus, all functions which must be performed on the master file will be placed in the same program. (perhaps people shouldn't create such a system, but they do).

I am not talking today about systems design, but about programming. Perhaps I should explain what we take this to mean in a commercial environment. We distinguish four levels in the construction of a complete system: specifying user facilities, defining the computer model, program design, and program coding and testing. Firstly, we are concerned with what the user expects to put into and get out of the system. Secondly, we determine what programs and files will be needed in order to achieve this. Next we decompose the programs into pieces or modules, and finally we must decide how to code each module. The top two levels are the function of the system designer (usually misnamed 'systems analyst'), the bottom two, of the programmer.

The interface between the systems analyst and the programmer is a program description. Typically, the analyst writes the description in

some fuzzy sort of procedural language. No compiler exists for this language (probably none could be written) so it is the job of the programmer to take this 'program' and translate it into a language for which there is a compiler available. Now the programmer reasons (usually correctly) that the description he is given contains errors. He therefore sees his role as follows: 'My job is to take this wrong procedure, determine what problem it is supposed to solve, and then solve that problem'.

4. Modularity

We need design tools. The flow chart is a hopeless tool for the program designer because it doesn't express structure. Figure 1, on the other hand, shows the hierarchical structure of a set of procedures without attempting to detail their flow of control. Arrows represent subroutine calls from one module to another. The goal is to break up a program into a number of such modules which can be constructed independently by several programmers. This has a number of advantages in project control, system maintenance, training, and especially in testing, but it does require that the modules be independently compilable.

(Q. At this point Professor Dijkstra asked: 'Why independent compilation? I can see the requirement that modules be independently written, but why must they be independently compiled?'
A. Systems contain very many modules - hundreds, or even thousands, in one system. It becomes very difficult and expensive to co-ordinate program compilations which draw on the current work of many programmers.)

What is a module? At Hoskyns the belief is that the appropriate unit is the pure procedure (one with no internal states). A stack is used to hold return links and procedure status and to provide working storage. (Recursion is thus allowed for, although seldom explicitly used.) How are such modules constructed? COBOL is too restrictive and insufficiently flexible, while the assembler permits us to build all the necessary features. I will, therefore, talk in terms of assembler language. We start from a set of S/360 macros. A module is invoked (and the necessary stack adjustments called for) and defined using the OBEY and PROC macros respectively.

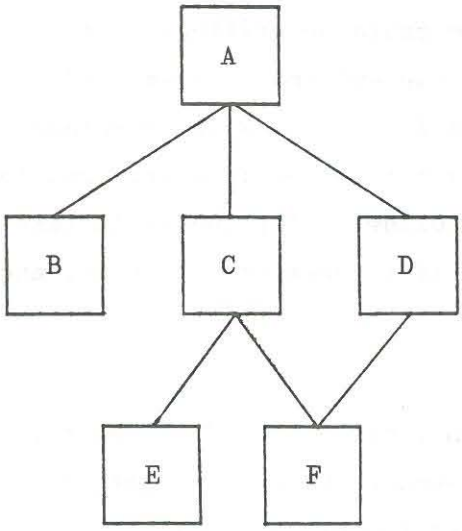

Figure 1: Hierarchical Structure

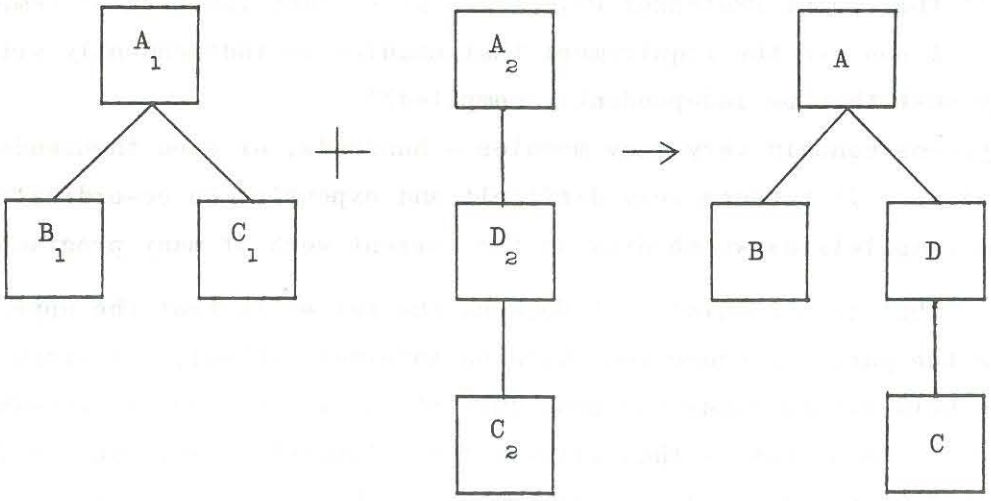

Figure 2: Compounding Data Structures

For example:

```
        OBEY PROCA, (P0,...,P4)
PROCA   PROC (D0,...,D4)
          :
PROCA   ENDS
```

5. Program design

This organisation is fine so far as it goes. Let us now turn our attention to the problem of program design, i.e. how to draw the hierarchical module chart. We start by recognising (as we have already done) that files have structures above the record level, that is, they are segmented, non-homogeneous structures. This structure in the data provides an initial organization for the modules. Thus a picture of the file is taken as the picture of the program. Of course, a given program will deal with a number of files, so we have a preliminary step of compounding these files into a single structure which is then used as the starting point for the modular structure. (See Figure 2).

We then allocate each of the program functions to a module of this structure. There are two basic principles of 'rightness' for allocating functions within the module structure: the module must be executed the right number of times, and it must be executed at the right time, for the particular function under consideration. The final step is to recognise that some boxes may have a low density of functions, and can be compounded into fewer (and larger) modules.

Because the file structures have to serve the needs of several programs, we may find at the preliminary design stage (of compounding file structures) that we have an impossible task: the file structures within one program may be incompatible. However, we choose to draw the module hierarchy, there will be modules whose relative levels are inverted with respect to the structure of one or more files.

A tool which has been found useful is a data structure called a parameter set which consists of a procedure name and a set of default parameters, and is set up by a PARSE macro. (The choice of the name PARSE causes no confusion, since commercial people don't know about parsing.) A parameter set allows a default procedure name and/or set of parameter values to be passed into a procedure. For example, in solving the eight queens problem discussed by Dijkstra, it is appropriate to set up the parameters of the print routine at the top level (Figure 3), but actually to invoke it only when a complete solution is found.

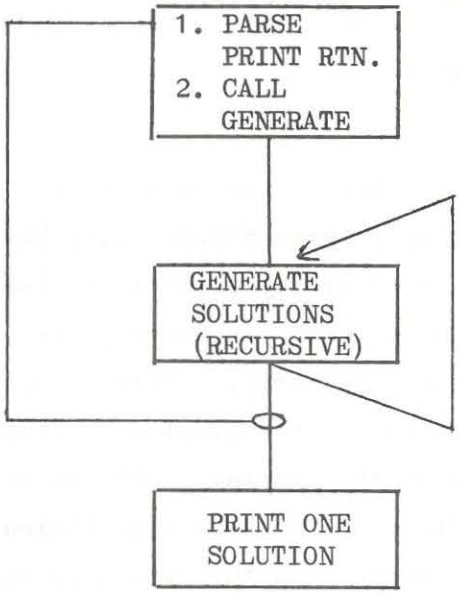

Figure 3: Use of PARSE in '8 Queens' Problem

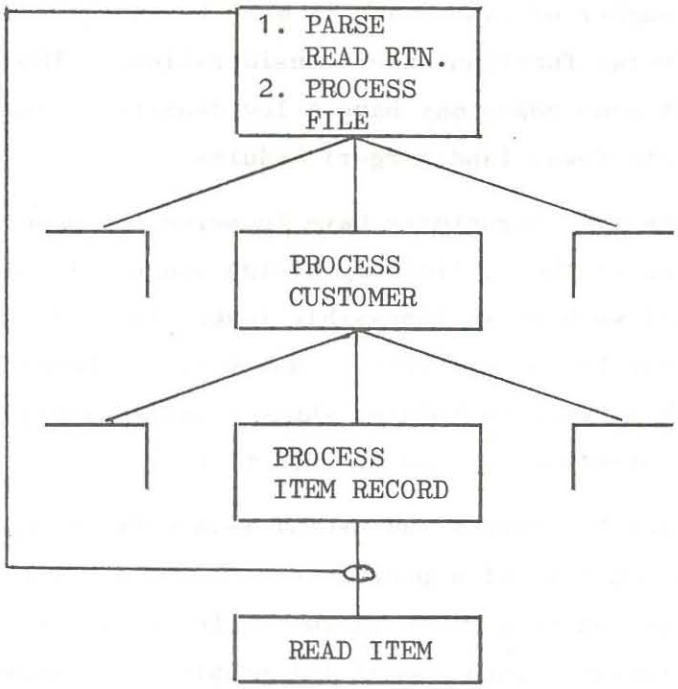

Figure 4: Use of PARSE for File-independence

A second application is shown in Figure 4. The program is to be written in a file-independent form. Where should the item records be read? Obviously, this should be done in the item routine, but it is not desirable to reference the file explicitly at this level. Therefore, the file access procedure is passed via a parameter set.

Similarly, problems may be resolved where the file structure does not match the logical structure of the routines (e.g. in preparing an invoice, the page heading will occur in the file above the line items, but whenever a line causes a page to overflow the heading routine - passed in by a parameter set - must be called as a lower-level procedure).

Another use comes in data validation, where errors may be detected at many different levels, but a common error recovery procedure is to be used. We may PARSE a call of a system procedure which unwinds the stack to the level at which the PARSE has been executed and returns to a point in the procedure at that level.

6. Structure within modules

The macros discussed so far have served to clarify and improve the structure of a program constructed of modules, but there still remains the problem of a free-for-all within a module. This is undesirable, thus leading us to the development of macros for structuring modules internally. In particular, we have found that GO TO's lead to an inordinate number of bugs, crossed loops, etc., so several control macros have been introduced to replace them with more nicely structured forms. DOGRP ... ENDS serves to group a set of statements as a subroutine (without implying iteration). IF ... ELSE ... ENDS provides the customary conditional statement, and LOOP ... ENDS specifies iteration. These structures all nest within themselves and each other.

These macros greatly assist the structuring of modules, but there are still situations where it is hard to cope. For example, we may often wish to exit from a loop at some point other than its final instruction. The QUIT macro can be used for this purpose. In a more complex situation, suppose that there are two conditions VALID1 and VALID2 and five procedures P1, ..., P5. P1 computes VALID1 and P2 computes VALID2. If both VALID1 and VALID2 are true it is required to invoke P1, ..., P4, otherwise to invoke P5. This can be done using DOGRP and IF:

```
A     DOGRP
      P1
B     IF VALID1
      P2
C     IF VALID2
      P3
      P4
C     ELSE
      P5
C     ENDS
B     ELSE
      P5
B     ENDS
A     ENDS
```

This solution is cumbersome and inelegant because a. the invocation of P5 must be repeated, and b. it requires a nesting structure three deep for a computational structure which is clearly not three deep. The POSIT macro avoids both these objections. The clause following POSIT is executed until a successful QUIT, (or the end) is found. The clause following ADMIT is executed following any successful QUIT, but not otherwise.

Repeating the previous example:

```
A     POSIT VALID1,2
      P1
      QUIT A ¬VALID1
      P2
      QUIT A ¬VALID2
      P3
      P4
A     ADMIT INVALID
      P5
A     ENDS
```

(Q: 'Why is it necessary to label the ENDS and ELSE macros if these structures properly nest?'
A: 'In the simple case they are not needed and a default label is supplied. However, (as in PL/1) a single labelled ENDS can be used to close all levels up to and including the one with the specified label.')

POSITs can be passed as parameters. One could be used in the eight queens problem (POSITing failure, and ADMITing success by executing a QUIT). There are many other such applications.

7. Generalisation

Since our design method gives a module hierarchy largely determined by the file structures, we often find that programs containing different functions share a similar or identical high-level structure. It is possible to regard a program as a file processor, determined by (and generable from) the file structures, together with a number of functions and a scheduler to select the functions to be activated at each point in the file processing. Something like this is the basis of RPG (Report Program Generator) systems.

8. Summary

The commercial programming environment is not technically very exciting. It can be a struggle even to persuade people that a pure procedure is possible (they don't understand the stack). Most commercial data processing installations are in a shambles. They are suspicious of academics and don't read the literature. The manager has a big office with the schedule on the wall (that's why he needs a big office). Programmers come in and report the bad news, and he sits there and takes it on the chin. Most programmers are far too 'helpful' when schedules are made up — they will agree to deliver in whatever time the manager asks. Unfortunately, however, most programmers overestimate their capacity.

'They told him it couldn't be done.
He smiled and went straight to it.
He tackled that program that couldn't be done,
And he couldn't do it.'

(Q: (Professor Michaelson) 'What attitude should we convey to our students who will be entering the commercial programming environment?'
A: 'Your current university courses seem to me pretty good; they cover the fundamentals. For example, a course in ALGOL conveys an understanding of the most important principles. What the student needs in addition is experience with certain types of problems. Somewhat like natural languages, these problems are not readily reducible to mathematics. What he needs is a systematic way of thinking about that type of problem.')