JACKSON ON JACKSON

Report on Infotech Course SG6: Structured Program Design
: DP Management Assessment

Date of Course 28th-29th October 1976

1. Introduction

This report is in three parts. The first is an account of the course, the second is a summary of the Michael Jackson technique, and the third is my conclusion on the method, and my recommendations for LOLA.

2. The Course

2.1 The lecturer - Michael Slavin

Mr. Slavin's name may be familiar - he was a councillor for the borough of Greenwich back in the 60's, and had an interest in the old LEO III at John Humphreys House. He still follows the progress of the old consortium, and wondered why the SE system had collapsed whilst LOLA was successful. I explained the different paths taken by LOLA.

Mr. Slavin has had a fairly wide range of jobs within the computing field, and in fact worked with Michael Jackson for Hoskyns Systems Research from 1966-1970, before rejoining him in 1975.

He tried to project the image of cool efficiency - "..... the time is now 10.15 and 32 seconds - we will break for coffee and be back by 10.30 and 32 seconds exactly". This largely fell flat on his audience of hardened professionals. I felt his manner was rather misplaced for the audience. He tended to lecture down rather than to.

2.2 The Audience

Of the 24 people on the course, 11 had come from Europe. We all had to introduce ourselves and say why we were on the course. The majority of people were managers, and only about a quarter were directly involved with programming. The others were project managers and computer managers. Most people were IBM users and COBOL and PL/1 were the most frequently used languages. The main reason for coming on the course was to gain a feel for the Jackson technique and to compare it with other IPT methods.

2.3 Session 1

After the introductions, the lecturer told us the history of computers from Babbage to the present day. From this he went on to discuss the roles of analysts and programmers. He emphasised that systems analysts were really business analysts or systems designers. Program design is better left to the programmers - ".....analysts must not wear two hats".

2.3 cont'd

The various aspects of a programmer's job were mentioned - the maintenance role, being the modification of the original program, and development work. As systems became more complex, so did the management of projects. The concept of modular programming evolved - a kind of production engineering. Program writing became an art, and the lecturer likened programmers to the medieval theologians, having endless arguments over the smallest points.

This session was quite interesting, but not really relevant. If it was meant as an introduction it went on too long.

2.4 Session 2

For the first time the word STRUCTURED was mentioned. The lecturer made the point that all programs were structured - some had a good structure others a bad one. He then went through the different papers published on Structured Programming. Dijkstra (the "goto" statement considered harmful), Bohm & Jacopini (the "goto" statement unnecessary), Nicklaus and Wirth (stepwise refinement) and the various IBM publications on Improved Programming Techniques by Mills, Constantine, Myers and Yourdon. The lecturer seemed very sceptical of IBM methods - he explained them badly and dismissed them. I was amazed how he glossed over HIPO. All there methods, he said, were aimed at the presentation of the program, and failed to address the fundamental problem of design. The key to it all is the data structure. A correct program structure is one that reflects accurately the form of the applications problem. This is the basis of Jackson's method and similar theories have been put forward by Wamier.

This session seemed to be continuing the introduction, and was frustrating. Possibly the comparison between Jackson and other methods could have come at the end of the course.

2.5 Session 3

The three basic components of a program were defined: sequence, iteration and selection. It was shown that these also exist in data structures. We then went on to a lengthy digression about representing "selection" in a program. The faults in commonly used programming languages were pointed out, even JCL was attacked. The constrictions of the languages have come to dominate design, but as far as the lecturer was concerned coding is unimportant, and to illustrate this, he produced example of bad coding which was neatly laid out within the IF-THEN-ELSE format, and "correct" code containing many GOTOs and labels without any indentation. This was immediately seized on by many of us, including myself, and we wasted a lot of time since it was not really a relevant topic.

This session started well, but went off course.

2.6 Session 4

We returned after tea to the subject, and got on to the relation between data structures and program structures: this must be one for one in the same order. So,we have the design problem of decomposing the task into elementary operations of the programming language, and arranging these operations in a suitable structure so that they will be executed in the right order with right relationships one with another. This leads on to the design procedure of defining data structures, forming the program structure and listing and allocating elementary operations.

We went over several simple examples.

This session was the first one which kept to the subject and was useful.

2.7 Session 5

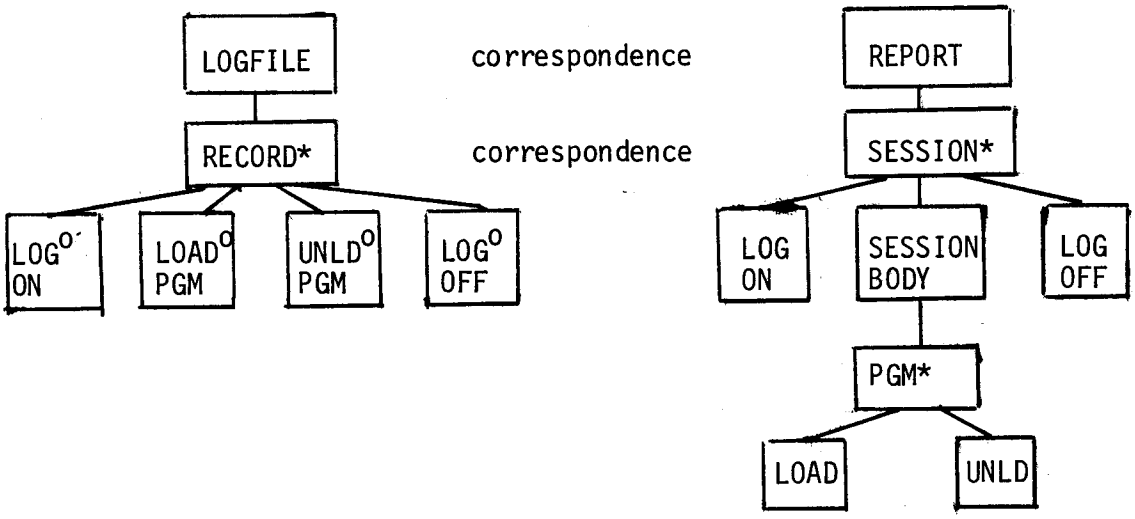
The more complicated forms of programs were now mentioned. The concept of 'promoted read' were introduced. Quite often, the processing of a record can be affected by its position in a file, and so the reading of the next record before completing the processing of the current record can be useful. This is used in practise by reading a record before entering an iterative loop, and reading the next record before the end of the loop. Structure clashes exist when a program has to handle two or more data structures and it is impossible for the same program structure to correspond perfectly with all of the data structures simultaneously. The different types of clashes were named: ordering clash, boundary clash and interleaving clash, and examples of each were given.

An ordering clash occurs when the output is to be presented in a different order from the input. For example, if we have a matrix which is held on a deck of cards, with each card containing the elements of one column of the matrix, and we wish to point it out row by row. There is a good correspondence between input matrix and output matrix, but none between column and row nor column element and row element.

A boundary clash occurs when there is a difference in synchronisation between data structures. For example, if we have a paper tape file containing telegram messages, with the data being held in blocks, and wish to separate the messages. There is a good correspondence between input file and output file, but none between input block size and output block size, but there is a good correspondence between words within blocks and characters within words.

2.7 cont'd

An interleaving clash occurs when there is a difference between the hierarchical structure of the files. For example, if we have a log file containing details of the usage of a time-sharing system and we wish to analyse this and produce reports by sessions, then the structures will be as shown.



* = repeat

o = either/or

This session was very useful, and we were now finding out what we came on the course for.

2.8 Session 6

We continued to discuss structure clashes, and got on to more complicated examples. The idea of backtracking was put forward as a solution to the kind of problems encountered during vetting, when several fields have to be checked before the data as a whole can be processed. The POSIT,QUIT and ADMIT method was explained: we assume the data is correct, but build in escape routes if errors are found - these are then dealt with in the error routines.

Structure clashes were summarised as being resolvable by breaking down the problem into one or more simple routines, each of which could be dealt with in a "correct" program, but one with a very peculiar format.

This session was a continuation of the previous one.

2.9 Session 7

Program inversion. By breaking down a program into several parts, there is the problem of maintaining a continuous flow. The Jackson solution is to produce a sub-program for each file where there are data clashes. If one of these is defined as the main program, then the others can be called when required. A simple example using toy bricks: to change a boat into a castle one way is to have two stages - stage one to breakup the boat into single bricks, stage two to build the castle from the pile of bricks. Combined into one, instead of stock piling the bricks, the castle can be built with the bricks as they are dismantled. Now each of the programs can be considered as the main program: using the example, the dismantling can be the main program calling the constructions routine whenever a brick is available, or the construction can be the main program, calling the dismantling routine whenever a brick is required.

A programming example is the telegram problem referred to in 2.7. The first step is to consider an intermediate file of words, produced from the input blocked format. This file can then be rebuilt into telegrams by a second step. The similarity with the bricks problem can now be seen.

Mention was then made of timescales of program development using these new techniques. These worked out at 90% design and 10% for coding and testing. Once a program had been designed correctly the coding became simply a clerical task and the testing almost unnecessary, since the correctness had been 'proved' by the very design of the program. A lot of the audience found this hard to swallow.

This session covered the final parts of the technique, albeit very briefly. The thinking on testing did seem to be rather superficial.

2.10 Session 8

A review of the course was made and questions answered. The general opinion seemed to be that too little time had been spent on the method itself - only half of the time had been taken up with explanations of how the Michael Jackson technique worked. I felt that by emphasising that this was the only correct method of program design, and that all other methods should be dismissed, the lecturer lost credibility - he oversold his product.

This session really summed up the course: perhaps I expected too much from it. However, it appears to me that one day would be long enough to cover the subject superficially, as on this course, but that given two days more time could be spent on the subject.

3. Summary of the Michael Jackson Technique

The all-important objective is correct structure. The structure is hierarchical, each component can be sequence, iteration selection or elementary.

The program structure must be based on data structure. The design procedure is to define the data structures and form the program structure by correspondences. The elementary operations can then be listed, and these are then allocated to program components, ensuring that the allocation is kept simple - if not, then the program structure will be wrong. The coding can then commence, and the resultant program has guaranteed context for each component - data to be processed is always present, and the required conditions are always obtained.

To resolve structure clashes, the program is split into two or more programs and recombined using the program inversion technique.

Backtracking is used when assumptions have to be made in advance, with built-in escape routines when the assumptions turn out to be false.

The net result is a "correct" solution to the problem.

4. Conclusion

From the number of satisfied users, there is no doubt that the Jackson method does work. However, it does have several draw-backs. The major one appears to be one of the features upon which it is sold: it is very rigid, and thus cannot be gradually phased in. It is all or nothing. From this it follows that all the people involved in an application using the technique need to be fully conversant with the methods. There is quite a lengthy learning period, for although it is basically a simple system the mechanics of backtracking and program inversion are somewhat complicated and take time to grasp. The net result, however, is a set of programs which will all be very similar, thus making making maintenance simpler, since if you are conversant with one you know them all. But this seems hard to swallow. I do not think that an integrated data base system such as ours is the right vehicle for the method.

A further factor is our present involvement with IPT. If we went for Jackson then much of this work would need to be scrapped as it is incompatible.

The concept of "top down" does not figure in the technique, since the complete system is designed and then coded. There appears to be no consideration given to testing methods.

Certain aspects of the method are very interesting, and IBM are currently investigating it, and indeed some of their departments do use the technique.

4. cont'd

It is well worth keeping a close eye on further developments, as a modification of the method may well be useful as a design tool even if it cannot be entertained in its present form.

Note For more detailed explanations of the technique, refer to the Michael Jackson file, which is to be found in Projects 1.

Martin Jackson