Techniques of program design

A. D. WOODALL
Department of Computing, North Staffordshire Polytechnic, Blackheath Lane, Stafford

Two modifications to Michael Jackson's program design method are presented.

Received June 1985

1. INTRODUCTION

The fundamental principle of JSP, the program design method introduced by Michael Jackson, is the observation that if the structure of the data to be processed is exhibited in the form in which it is to be processed, then a program structure in accord with the data structures will be especially effective.

The present paper introduces two modifications to JSP which have been described more fully, together with illustrative examples, elsewhere.

2. DATA SUBSTRUCTURES

One element not widely used in JSP is a program subroutine, and it is not usual to identify any data structural element analogous to a subroutine. In fact it often pays to do so when we are dealing with problems whose solution can be thought of hierarchically: an inner problem which can be solved simply and completely by a JSP design, and an outer problem which is structurally simple, provided the inner can be thought of as a unit.

In order to bring this within the usual JSP formalism it helps to introduce a different element into the diagrammatic representation of both data and program structures. A suitable element is a box with vertical lines down the sides, like the representation of a subroutine in flow-chart symbolism.

In the previously cited paper, two examples of the use of this method are described in detail: they are outlined here.

2.1. A structure clash example

The problem is to input a file containing names and addresses to print on labels. There are exactly five lines in each label record, and the output must print labels side by side with three to a row, and with four rows to a page. Clearly there is a structure clash.

In this case the inner problem is chosen as the problem of dealing with an input comprising exactly three labels (or less). The clash is still there, but its resolution by way of an intermediate structure is obvious. The intermediate structure to be used is simply a table of 5 by 3 line segments, and since this can be accessed either by row or by column, it can be matched with both the input and the output.

Once this inner program has been written, the solution to the full problem is simple. It simply involves iteratively calling the inner program as a subroutine, and the requirement to cope with pages presents no particular problem.

2.2. Serial update

Substructures have also proved useful in situations where no structure clash is found and a solution is possible using the standard approach. The substructure method sometimes yields a simple solution more directly.

One such application is the serial file update problem, which has been extensively studied, see for example an paper by Dwyer, where a survey of solutions is given. A new treatment of serial file update using data substructures will be found in the previously cited paper by the present author, the solution comparing favourably with that presented by Dwyer. The inner problem in this case deals with an old master file with a single record, and a transaction file containing records with keys all less than or equal to the key of the old master file record.

3. PROBLEM TRANSFORMATION

3.1. Method

In a previous paper, a program was designed as a solution to a problem given in Jackson's book. This is the telegram program, and the method of design that was used differed considerably from that recommended by Jackson, leading to a simpler program.

The method in question has since been found to be applicable to many other program design problems. As an example the system-log problem, which is also dealt with in Jackson's book, has a simple solution when treated in this way (see the above-cited paper).

In essence the method demands that the outer system, of which the program is to be a part, should be ignored. Instead, the actual processing of data which is demanded of the program is thought of as if it were a blind procedure, and the simplest way of performing the necessary tasks is planned. In this sense, blind is meant to imply that the data are processed without any thought about their meaning or antecedents.

In the system-log problem, for example, input messages about jobs run on a batch system form the input: there are messages for the start of each job, the start of each program within a job, and for the end of each program and each job. A proper use of JSP demands that the program should have a component corresponding to 'job', and one corresponding to 'program'. The present proposal is to relax this requirement, defining the processing to be done in terms of counters to be kept as the input is read, and a report of the final state of the counters to be printed when input has terminated. The advantage gained is simplicity. The JSP solution involves resolution of a structure clash, problem transformation does not.
3.2. The dangers of problem transformation

It seems that there are two distinct fundamental features of JSP. The first is that the view adopted of the data to be processed by the program under design should be the same as that of the outer system of which the program is a part. The second is that the structure of the program itself should be consonant with that of the data it processes.

Clearly the second of these fundamentals is inviolable. Unless the program structure is made the same as the structure of the data being processed, there is nothing left of the method.

The first of the fundamentals is different. Other things being equal, it is highly desirable: for an obvious reason.

Future changes to the task the program has to perform may well relate to its outer structure. In the system-log problem, we may expect that amendments needed to the program will relate to programs or to jobs. If there is no component in our program identified with a job, an amended requirement referring to jobs may entail a fundamental redesign – or even worse a temptation to patch the program in a way that quite destroys its structure.

However, if we choose to disregard this requirement, JSP can be used to provide a good and simple program design when the data structures considered have been those actually handled by the program, unconstrained by insistence on their representation matching the outer system.

The advantage we have gained is simplicity of the program: it must be weighed against the disadvantages that may accrue during future maintenance. My own view is that the disadvantages can be overemphasised. Changes to the outer system that cannot be assimilated as simple changes to the data-processing tasks required will often imply changes to the data structures involved anyway, leading to fundamental redesign.

In some ways the situation within the overall system implementation context is a 'back-track' problem. We 'posit' (in the language of JSP) that no major redesign will become necessary. If it does we must accept the consequences.

It is worth emphasising that where complying with the outer system’s structure does not render the program design task more difficult, it should be done.

4. CONCLUSION

The usefulness of a program design methodology can perhaps be assessed by its popularity. By this test, JSP is clearly effective. However, that is not to say that the method should be regarded as inviolable.

The effectiveness of the changes proposed here is illustrated by the examples given, which are more fully expounded elsewhere.3 The wisdom of the second change – problem transformation – needs to be assessed in any particular context. It may be that it will be found useful for those areas of program design where JSP has so far made little impact – e.g. scientific and technical programming – but regarded as dangerous in the field of commercial data processing where JSP has found most application.

REFERENCES