# JSP In Perspective

## INTRODUCTION

JSP is a method of program design. Its origins lie in the data processing systems that grew up in the 1960s, when reliable, relatively cheap, and adequately powerful computers first became generally available. The fundamental abstraction in JSP is the sequential data stream. Originally, this abstraction was inspired and motivated by the sequential tape files that characterised data processing in the 1960s, but it quickly became clear that it had a much wider applicability. Today the JSP design method is valuable for such applications as embedded software and handling network protocols.

JSP arose from efforts by a small group of people in a data processing consultancy company to improve their programming practices, and to make their programs more reliable and easier to understand and to modify. In 1971 it became the core product of a very small new company, Michael Jackson Systems Limited, which offered development services, training courses, consultancy, and — from 1975 — software to support JSP design of COBOL programs. The name JSP — 'Jackson Structured Programming' — was coined by the company's Swedish licensee in 1974. In the commercial world, IBM had appropriated the name 'Structured Programming' in the early 1970s, and Yourdon Inc started offering courses in 'Structured Design' around 1974. A distinctive name was a commercial necessity. It was also technically appropriate to choose a distinctive and proprietary name: the JSP method was very different from its competitors.

## 1960S DATA PROCESSING SYSTEMS

Data processing systems of the early and middle 1960s were chiefly concerned with the processing of sequential files held on magnetic tape. Reliable tape drives had become widely available and commonly used in the late 1950s; exchangeable disk drives first became available when IBM introduced the 1311 drive in 1963. Disk was a very limited and expensive medium compared to tape. At 1965 prices a 1311 disk pack cost about £200 and held 2 million characters; a 2400-foot tape reel cost about £7 and held between 20 million and 60 million characters. For large files, which might contain millions of records, tape was the only realistic choice. Most data processing systems had large files.

Because tape is an inherently sequential medium, updating a single record of a master file could be done only by reading the whole file and copying it, updated, to a new tape. This very slow process was economical only if many records were to be updated, so tape systems were almost inevitably batch systems. Transactions — for example, payments received — were recorded daily or weekly on a transaction tape file. The transaction file was then sorted into the same sequence as the file of master records — for example, customer accounts — to which the transactions were to be

applied; it was then used in a batch update program to produce a new version of the master file whose records reflected the effects of the transactions.

It was always necessary to process the whole master file and to produce a complete new version, even if the batch contained transactions for very few master records. Processing a file that occupied one full tape might take an hour or more; some master files occupied dozens of tapes. Even worse, there might be several master files to be processed — for example, customers, orders, invoices, and products. The transaction file would then be sorted successively into the different sequences of the different master files, executing a batch update program for each master file and carrying partial results forward to the next update program in a transfer file that would also require to be sorted. To minimise processing time master files were amalgamated where possible. For example, the orders, instead of being held in a master file of their own, might be held in the customer master file, the order records for each customer following the customer record in the combined file. These choices resulted in a database with a hierarchical structure, held on magnetic tape: this was the kind of database for which IBM's database management system IMS was originally designed around 1966 in cooperation with North American Rockwell [Blackman 98].

## THE BASIC JSP IDEA

A common design fault in batch update programs was failure to ensure that the program kept the files correctly synchronised as it traversed their hierarchical structures. A read operation performed at the wrong point in program execution might read beyond the record to which the next transaction should be applied. The result would be erroneous processing of that transaction and, often, of the following transactions and master records. Another common design fault was failure to take account of empty sets — for example, of a customer with no outstanding orders. How could one design a program that would not have such faults?

In commercial and industrial programming in the 1960s, the program design question was chiefly posed in terms of 'modular programming': What was the best decomposition for each particular program? The primary focus was on the decomposition structure, not on encapsulation. A 1968 conference [Barnett 68] dedicated to modular programming attracted the participation of George Mealy, the computer scientist who gave his name to Mealy machines. The Structured Design ideas of coupling and cohesion [Stevens 74, Myers 76] took shape as an approach to modularity: it was claimed that a good design could be achieved by ensuring that the modules have high cohesion and low coupling.

The fundamental idea of JSP was very different: program structure should be dictated by the structure of its input and output data streams [Jackson 75]. If one of the sequential files processed by the program consisted of customer groups, each group consisting of a customer record followed by some number of order records, each of which is either a simple order or an urgent order, then the program should have the same structure: it should have a program part that processes the file, with a subpart to process each customer group, and that subpart should itself have one subpart that processes the customer record, and so on. The execution sequence of

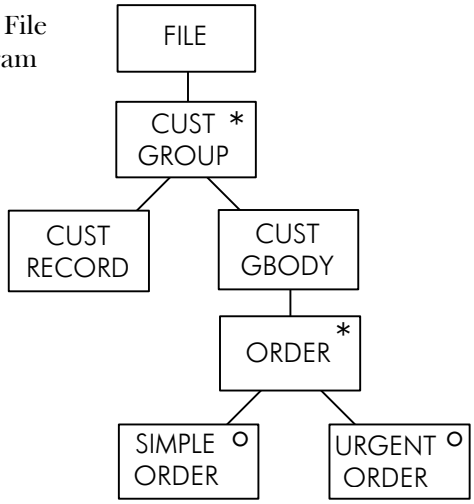
the parts should mirror the sequence of records and record groups in the file. Program parts could be very small and were not, in general, separately compiled.

The resulting structure can be represented in a JSP structure diagram, as in Figure 1. The diagram is a tree representation of the regular expression

( CustRecord ( SimpleOrder | UrgentOrder ) * ) *

in which the expression and all of its all subexpressions are labelled. Iteration is shown by the star in the iterated subexpression; selection is shown by the circle in each alternative.

Figure 1: Structure of a File and of a Program




The structure is simultaneously the structure of the file and the structure of a program to process the file. As a data structure it may be verbalised like this:

> "The *File* consists of zero or more *Customer Groups*. Each *Customer Group* consists of a *Customer Record* followed by a *Customer Group Body*. Each *Customer Group Body* consists of zero or more *Orders*. Each *Order* is either a *Simple Order* or an *Urgent Order*"

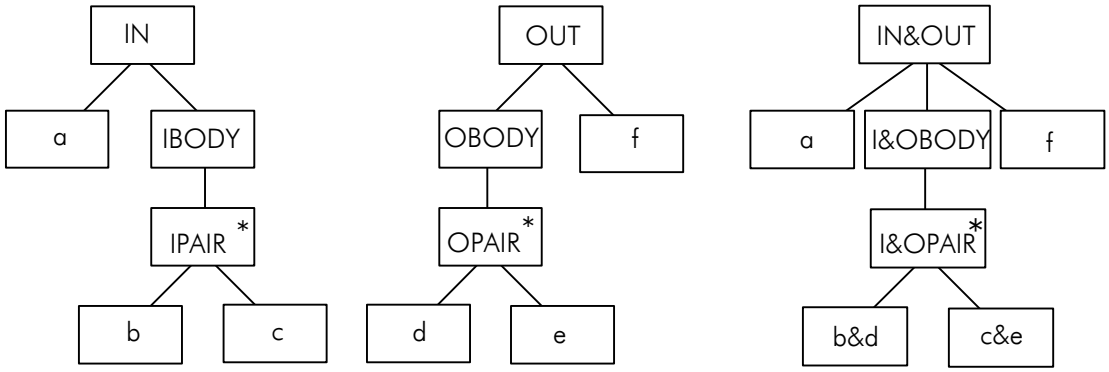
As a program structure it may be understood to mean:

```
program =
    { /* process file */
        while (another customer group) do
        { /* process customer group */
            process customer record;
            { /* process customer group body /*
                while (another order) do
                { /* process order */
                    if simple order then
                    { /* process simple order */ }
                    else
                    { /* process urgent order */ }
    }   }   }   }
```

## MULTIPLE STREAMS

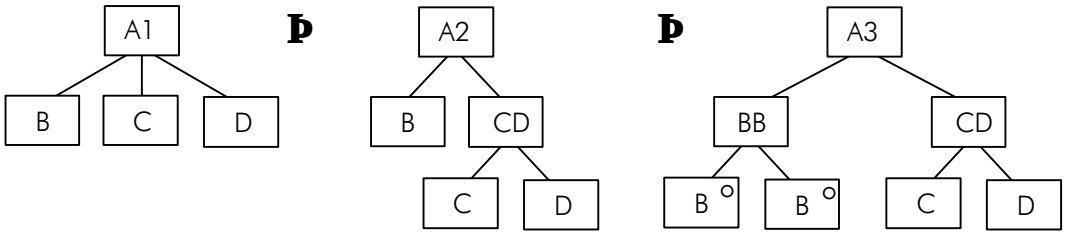
The JSP program design method insisted that the program structure should reflect all the stream data structures, not just one. Its first steps, then, are to identify the data structure of each file processed by the program, and to form a program structure that embodies them all. Such a program structure allows the designer to ensure easily that program execution will interleave all the file traversals correctly and will keep them appropriately synchronised. Figure 2 shows an example of a program structure based on two data structures.

Figure 2: Two File Structures and a Program Structure




For brevity, the example is stylised and trivial. The program processes an input file *IN* and and output file *OUT*. The successive *OPAIRs* of *OUT* are constructed from the successive *IPAIRs* of *IN*: that is, the *IPAIRs* and *OPAIRs* 'correspond functionally'. Similarly, the *d* and *e* records are computed from the *b* and *c* records respectively: that is, the *b* and *d* records correspond and the *c* and *e* records correspond. In this trivial example it is easily seen that the program structure embodies both of the file structures exactly. In more realistic examples, a program structure embodying all the file structures can be achieved by permissible rewritings of the file structures. Two such rewritings are shown in Figure 3.

Figure 3: Examples of Regular Expression Rewritings




The data structure A1 can be rewritten as A2, and A2 as A3. Permissible rewritings preserve the set of leaf sequences defined by the structure: A1, A2 and A3 all define the sequence <B,C,D>. They must also preserve the intermediate nodes of each structure: A2 may not be rewritten as A1, because the node CD would be lost. The eventual program structure must have at least one component corresponding to each component of each data structure.

## OPERATIONS AND CONDITIONS

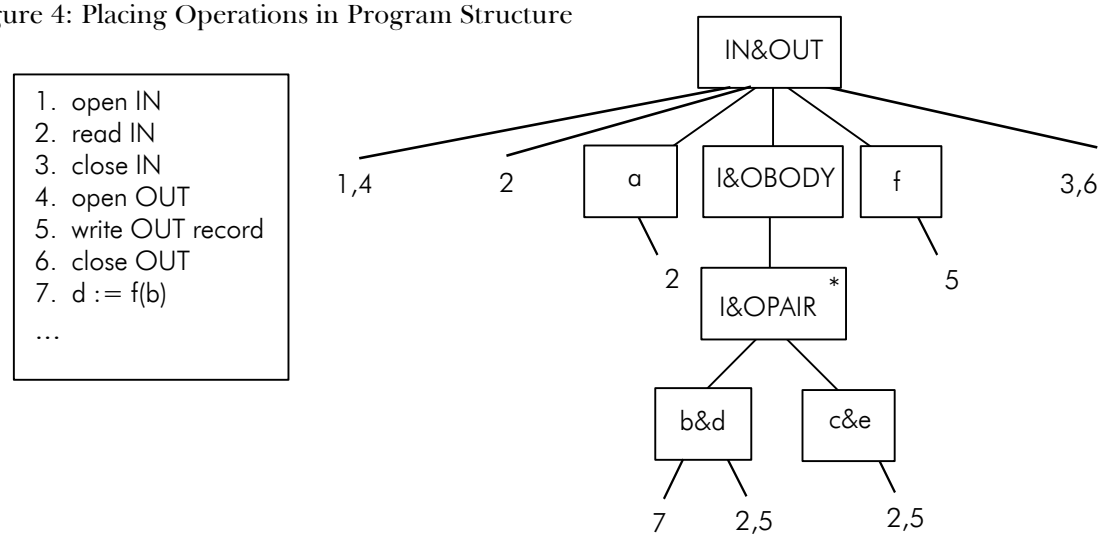
The prime advantage of a program structure that embodies all the file data structures, respecting the correspondences among their parts, is that it provides an obviously correct place in the program text for each operation that must be executed. It also clarifies the conditions needed as guards in iteration (loop) and selection (if-else or case) constructs.

The operations to be executed are file traversal operations, such as *open*, *close*, *read* and *write*, and other operations that compute output record values from values of input records. For example, in the trivial illustration of Figure 2 the operations may be:

*open IN, read IN, close IN, open OUT, write OUT record, close OUT, d := f(b),*

and so on. Each operation must appear in the program component that processes the operation's data. The *read* operations are a special case. Assuming that the input files can be parsed by looking ahead one record, there must be one *read* operation following the *open* at the beginning of the file, and one at the end of each component that completely processes one input record. So, for the example of Figure 2, the operations must be placed as shown in Figure 4.

Figure 4: Placing Operations in Program Structure




The correspondence of program and data structures, together with the scheme of looking ahead one record, makes it easy to determine the iteration and selection conditions. For example, the condition on the iteration component *I&OBODY* is

while (another *I&OPAIR*)

which translates readily into

while (*IN* record is *b*)

in which '*IN* record' refers to the record that has been read ahead and is currently in the *IN* buffer.

## DIFFICULTIES

The development procedures of a method should be closely matched to specific properties of the problems it can be used to solve. The development procedures of basic JSP, as they have been described here, require the problem to possess at least these two properties:

- the data structures of the input and output files, and the correspondences among their data components, are such that a single program structure can embody them all; and
- each input file can be unambiguously parsed by looking ahead just one record.

Absence of a necessary property is immediately recognisable by difficulty in completing a part of the JSP design procedure. If the file structures do not correspond appropriately it is impossible to design a correct program structure: this difficulty is called a *structure clash*. If an input file can not be parsed by single look ahead it is impossible to write all the necessary conditions on the program's iterations and selections: this is a *recognition difficulty*.

Although these difficulties are detected during the basic JSP design procedure, they do not indicate a limitation of JSP. They indicate inherent complications in the problem itself, that can not be ignored but must be dealt with somehow. In JSP they are dealt with by additional techniques within the JSP method.

## STRUCTURE CLASHES

There are three kinds of structure clash: *interleaving clash*, *ordering clash*, and *boundary clash*.

In an interleaving clash, data groups that occur sequentially in one structure correspond functionally to groups that are interleaved in another structure. For example, the input file of a program may consist of chronologically ordered records of calls made at a telephone exchange; the program must produce a printed output report of the same calls arranged chronologically within subscriber. The 'subscriber groups' that occur successively in the printed report are interleaved in the input file.

In an ordering clash, corresponding data item instances are differently ordered in two structures. For example, an input file contains the elements of a matrix in row order, and the required output file contains the same elements in column order.

In a boundary clash, two structures have corresponding elements occurring in the same order, but the elements are differently grouped in the two structures. The boundaries of the two groupings are not synchronised.

Boundary clashes are surprisingly common. Here are three well-known examples:

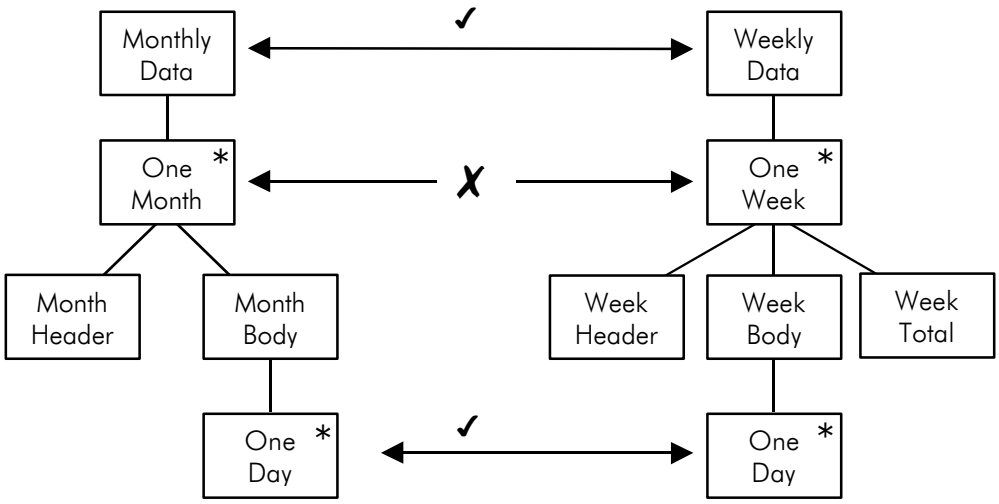
- The calendar consists of years, each year consisting of a number of days. In one structure the days may be grouped by months, but by weeks in another structure. There is a boundary clash here: the weeks and months can not be synchronised.

- A chapter of a printed book consists of text lines. In one structure the lines may be grouped by paragraphs, but in another structure by pages. There is a boundary clash because pages and paragraphs can not be synchronised.
- A file in a low-level file handling system consists of variable-length records, each consisting of between 2 and 2000 bytes. The records must be stored sequentially in fixed blocks of 512 bytes. There is a boundary clash here: the boundaries of the records can not be synchronised with the boundaries of the blocks.

The difficulty posed by a boundary clash is very real. The clash between weeks and months causes endless trouble in accounting: in 1923 the League of Nations set up a Special Committee of Enquiry into the Reform of the Calendar to determine whether the clash could be resolved by adopting a new calendar [Achelis 59]. The clash between records and blocks affected the original IBM OS/360 file-handling software: the 'access method' that could handle the clash — software that supported 'spanned records' — proved the hardest to design and was the last to be delivered.

The JSP technique for dealing with a structure clash is to decompose the original program into two or more programs communicating by intermediate data structures. A boundary clash, for example, requires a decomposition into two programs communicating by an intermediate sequential stream. The structure of the intermediate stream is based on the 'highest common factor' of the two clashing structures. For example, an accounting program may have a boundary clash between an input file structured by months and an output file structured by weeks. Figure 5 shows the data structures.

Figure 5: Structures Exhibiting a Boundary Clash



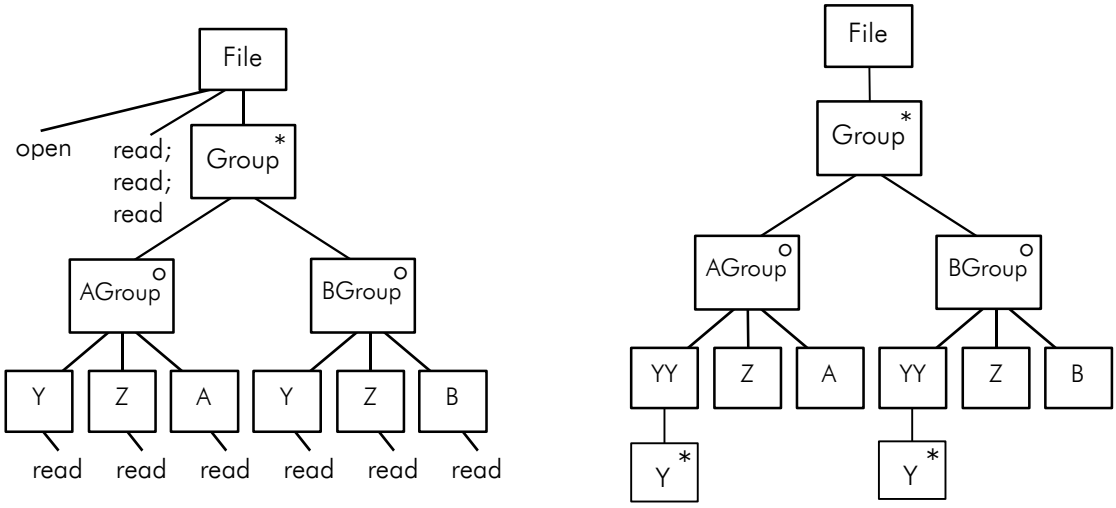

The solution to the difficulty is to decompose the program into two: one program handles the *Months*, producing an intermediate file that is input to a second program that handles the *Weeks*. For the second program the intermediate file structure must have no *Month* component: any necessary information from the *MonthHeader* record must therefore be encoded in the *OneDay* records of the intermediate file.

## RECOGNITION DIFFICULTIES

A recognition difficulty is present when an input file can not be unambiguously parsed by single look-ahead. Sometimes the difficulty can be overcome by looking ahead two or more records; sometimes a more powerful technique is necessary.

The two cases are illustrated in Figure 6. The structure on the left can be parsed by looking ahead three records: the beginning of an *AGroup* is recognised when the third of the lookahead records is an *A*. But the structure on the right can not be parsed by any fixed look-ahead. The JSP technique needed for this structure is *backtracking*.

Figure 6: Structures Requiring Multiple Read-ahead and Backtracking




The JSP procedure for the backtracking technique has three steps:

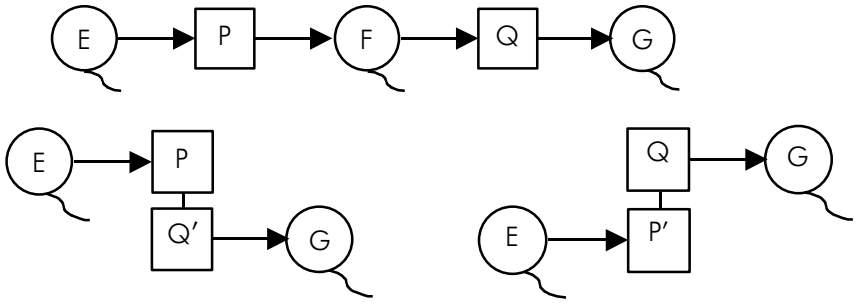
- First, the recognition difficulty is simply ignored. The program is designed, and the text for the *AGroup* and *BGroup* components is written, as usual. No condition is written on the *Group* selection component. The presence of the difficulty is marked only by using the keywords *posit* and *admit* in place of *if* and *else*.
- Second, a *quit* statement is inserted into the text of the *posit AGroup* component at each point at which it may be detected that the *Group* is, in fact, not an *AGroup*. In this example, the only such point is when the *B* record is encountered. The *quit* statement is a tightly constrained form of *GO TO*: its meaning is that execution of the *AGroup* component is abandoned and control jumps to the beginning of the *admit BGroup* component.
- Third, the program text is modified to take account of side-effects: that is, of the side-effects of operations executed in *AGroup* before detecting that the *Group* was in fact a *BGroup*.

## PROGRAM INVERSION

The JSP solution to structure clash difficulties seems, at first sight, to add yet more sequential programs and intermediate files to systems already overburdened with time-consuming file processing. But JSP provides an implementation technique — *program inversion* — that both overcomes this obstacle and offers other important positive advantages.

The underlying idea of program inversion is that reading and writing sequential files on tape is only a specialised version of a more general form of communication. In the general form programs communicate by producing and consuming sequential streams of records, each stream being either unbuffered or buffered according to any of several possible regimes. The choice of buffering regime is, to a large extent, independent of the design of the communicating programs. But it is not independent of their scheduling. Figure 7 shows three possible implementations of the same system.

Figure 7: Using Inversion to Eliminate an Intermediate File




In the upper diagram two 'main' programs, P and Q, communicate by writing and reading an intermediate tape file F. First P is run to completion; then F is rewound; then Q is run to completion. In the lower left diagram the intermediate tape file has been eliminated: the 'main' program Q has been *inverted with respect to F*. This inversion converts Q into a subroutine Q′, invoked by P whenever P requires to produce a record of F. Q′ functions as a 'consume next F record' procedure. Similarly, in the lower right diagram the 'main' program P has been *inverted with respect to F*. This inversion converts P into a subroutine P′, invoked by Q whenever Q requires to consume a record of F. P′ functions as a 'produce next F record' procedure. Both inversions interleave execution of P and Q as tightly as possible: each F record is consumed as soon as it has been produced.

Two or more programs may be inverted in one system. Inverting Q with respect to F, and P with respect to E, gives a subroutine P″ that uses the lower-level subroutine Q′; the function of the two together is to consume the next record of E, producing whatever records of G can then be constructed.

Inversion has an important effect on the efficiency of the system. First, it eliminates the storage space and device costs of the intermediate tape file. It eliminates the time required by the device to write and read each record of F, and also the 'rewind time' to reposition the newly written file for reading: for a magnetic tape file this

may be many minutes for one reel. Second, it makes each successive record of G available with the least possible delay: each G record is produced as soon as P has consumed the relevant records of E.

Inversion also has an important effect on the program designer's view of a sequential program. The 'main program' P, the subroutine 'P inverted with respect to F', and the subroutine 'P inverted with respect to E', are seen as identical at the design level. This is a large economy of design effort. The JSP-COBOL preprocessor that supports JSP design for COBOL programs allows the three to differ only in their declarations of the files E and F.

The effect on the program designer's view goes deeper than an economy of design effort. An important example of the distinction between design and implementation is clarified; procedures with internal state can be designed as if they were main programs processing sequential message streams; and JSP design becomes applicable to all kinds of interactive systems, and even to interrupt handlers. Program inversion also suggests the modelling of real-world entities by objects with time-ordered behaviours: this is the basis of the eventual enlargement of JSP to handle system specification and design [Jackson 83].

## A Perspective View of JSP

Although JSP was originally rooted in mainframe data processing, it has been applied effectively in many environments. For applying JSP, the necessary problem characteristic is the presence of at least one external sequential data stream, to provide a given data structure that can and should be used as the basis of the program structure. Many programs have this characteristic. For example:

- a program that processes a stream of EDI messages;
- an automobile cruise control program that responds to the changing car state and the driver's actions;
- a program that justifies text for printing;
- a file handler that responds to invoked operations on the file;
- a program that generates HTML pages from database queries.

JSP was developed in the commercial world, often in ignorance of work elsewhere. Some of the JSP ideas were reinventions of what was already known, while others anticipated later research results. The JSP relationship between data structures and program structure is essentially the relationship exploited in parsing by recursive descent [Aho 77]. Some of the early detailed JSP discussion of recognition difficulties dealt with aspects that were well known to researchers in formal languages and parsing techniques. The idea of program inversion is closely related to the Simula [Dahl 70] concept of semi-coroutines, and, of course, to the later Unix notion of pipes as a flexible implementation of sequential files. There was also one related program design approach in commercial use: the Warnier method [Warnier 74] based program structure on the regular data structure of one file. Program decomposition into sequential processes communicating by a coroutine-like

mechanism was discussed in a famous early paper [Conway 63]; it was also the basis of a little-known development method called Flow-Based Programming [Morrison 94].

The central virtues of JSP are two. First, it provides a strongly systematic and prescriptive method for a clearly defined class of problem. Essentially, independent JSP designers working on the same problem produce the same solution. Second, JSP keeps the program designer firmly in the world of static structures to the greatest extent possible. Only in the last step of the backtracking technique, when dealing with side-effects, is the JSP designer encouraged to consider the dynamic behaviour of the program. This restriction to designing in terms of static structures is a decisive contribution to program correctness for those problems to which JSP can be applied. It avoids the dynamic thinking — the mental stepping through the program execution — that has always proved so seductive and so fruitful a source of error.

## ACKNOWLEDGEMENTS

The foundations of JSP were laid in the years 1966-1970, when the author was working with Barry Dwyer, a colleague in John Hoskyns and Company, an English data processing consultancy. Many of the underlying ideas can be traced back to Barry Dwyer's contributions in those years. Refining the techniques, and making JSP more systematic and more teachable in commercial courses, was the work of the following four years in Michael Jackson Systems Limited.

The JSP-COBOL preprocessor was designed by the author and Brian Boulter, a colleague in Michael Jackson Systems Limited. Brian Boulter was responsible for most of the implementation.

Many other people contributed to JSP over the years — John Cameron, Tony Debling, Leif Ingevaldsson, Ashley McNeile, Hans Naegeli, Dick Nelson (who introduced the name 'JSP'), Bo Sanden, Peter Savage, Mike Slavin and many others. A partial bibliography of JSP can be found in [Jackson 94].

Daniel Jackson read a draft of this paper and made several improvements.

## REFERENCES

[Achelis 59] Elisabeth Achelis; *The Calendar for the Modern Age*; Thomas Nelson and Sons, 1959.

[Aho 77] A V Aho and J D Ullman; *Principles of Compiler Design*; Addison-Wesley, 1977.

[Barnett 68] Barnett and Constantine eds; *Modular Programming: Proceedings of a National Symposium*; Information & Systems Press, 1968.

[Blackman 98] K R Blackman; *IMS celebrates thirty years as an IBM product*; IBM Systems Journal, Volume 37 Number 4, 1998.

[Conway 63] Melvin E. Conway; *Design of a Separable Transition-Diagram Compiler*; Communications of the ACM Volume 6 Number 7, 1963.

[Dahl 70] O-J Dahl, B Myhrhaug and K Nygaard; *SIMULA-67 Common Base Language*. Technical Report Number S-22, Norwegian Computer Centre, Oslo, 1970.

[Jackson 75] M A Jackson; *Principles of Program Design*; Academic Press, 1975.

[Jackson 83] M A Jackson; *System Development*; Prentice-Hall International, 1983.
[Jackson 94] Michael Jackson; *Jackson Development Methods: JSP and JSD*; in Encyclopaedia of Software Engineering, John J Marciniak ed, Volume I; John Wiley and Sons, 1994.
[Morrison 94] J Paul Morrison; *Flow-Based Programming: A New Approach to Application Development*; Van Nostrand Reinhold, 1994.
[Myers 76] Glenford J. Myers; *Software Reliability: Principles & Practices*; Wiley, 1976.
[Stevens 74] W P Stevens, G J Myers, and L L Constantine; *Structured Design*; IBM Systems Journal Volume 13 Number 2, 1974.
[Warnier 74] Jean-Dominique Warnier; *Logical Construction of Programs*; H E Stenfert Kroese, 1974, and Van Nostrand Reinhold, 1976.

Michael Jackson
11th April 2001