A report on JSP by C.A.R. Hoare, 1977
A Tribute to Michael Jackson · Vancouver · May 2009
Daniel Jackson · MIT
'The stores section in a factory issues and receives parts. Each issue and each receipt is recorded on a punched card: the card contains the part-number, the movement type (I for issue, R for receipt) and the quantity. The cards have already been copied to magnetic tape and sorted into part-number order. The program to be written will produce a simple summary of the net movement of each part. The format of the summary is:

STORES MOVEMENTS SUMMARY

<table>
<thead>
<tr>
<th></th>
<th>NET MOVEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>A5/132</td>
<td>-450</td>
</tr>
<tr>
<td>A5/197</td>
<td>1760</td>
</tr>
<tr>
<td>B41/728</td>
<td>7</td>
</tr>
</tbody>
</table>

No attention need be paid to such refinements as skipping over the perforations at the end of each sheet of paper.'

The first step of the design procedure, the data step, is to draw data structures of all the files in the problem. The result of the data step is:
The Michael Jackson Design Technique

A study of the theory with applications.

C. A. R. Hoare

March 1977.

The execution of a computer program involves the execution of a series of elementary commands, and the evaluation of a series of elementary tests.*

It is in principle possible to get a computer to record each elementary command when it is executed, and also to record each test as it is evaluated.*
1 Programs, Traces and Regular Expressions

The execution of a computer program involves the execution of a series of elementary commands, and the evaluation of a series of elementary tests (which, in fact, the Michael Jackson technique tends to ignore). It is in principle possible to get a computer to record each elementary command when it is executed, and also to record each test as it is evaluated (together with an indication whether it was true or false). Such a record is known as a ‘trace’; and it can sometimes be helpful in program debugging.

It is obviously very important that a programmer should have an absolutely clear understanding of the relation between his program and any possible trace of its execution. Fortunately, in the case of a structured program, this relation is very simple – indeed, as Dijkstra pointed out, this is the main argument in favour of structured programming. The relation may be defined as follows:

(a) For an elementary condition or command (input, output, or assignment), the only possible trace is just a copy of the command. This is known as a ‘terminal symbol’, or ‘leaf’, of the structure tree.
(b) For a sequence of commands (say \( P ; Q \)), every possible trace consists of a trace of \( P \) followed by a trace of \( Q \) (and similarly for sequences longer than two).
(c) For a selection (say \( P \cup Q \)), every possible trace is either a trace of \( P \) or a trace of \( Q \) (and similarly for selections of more than two alternatives).
(d) For an iteration (say \( P^* \)), every possible trace is a sequence of none or more traces, each of which is a (possibly different) trace of \( P \). Zero repetitions will give rise to an ‘empty’ trace. Thus it can be seen that a program is a kind of regular expression, and that the set of possible traces of the program is the ‘language’ defined by that expression. In fact, the ‘language’ will contain many traces that can never be the result of program execution; but for the time being it will pay to ignore that fact.

Of course, a complete trace of every action of a program will often be too lengthy for practical use. We therefore need a selective trace, which is a record of all actions affecting some particular variable, or some particular input or output file. (Note that a complete trace is a merging of all the selective traces derived from it; but it is not possible to work out which merging it was from the selective traces alone.) A record of all input instructions on a particular input file will be effectively the same as a copy of the particular data presented to the program on that particular execution of it; and similarly a selective trace of output instructions will be nothing but a copy of the output file. For program debugging such traces would never be required. But the idea of the selective trace is the basis of the whole Michael Jackson Design Technique.
a JSP example
'The stores section in a factory issues and receives parts. Each issue and each receipt is recorded on a punched card: the card contains the part-number, the movement type (I for issue, R for receipt) and the quantity. The cards have already been copied to magnetic tape and sorted into part-number order. The program to be written will produce a simple summary of the net movement of each part. The format of the summary is:

STORES MOVEMENTS SUMMARY

<table>
<thead>
<tr>
<th>Item</th>
<th>Net Movement</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>A5/132</td>
<td>NET MOVEMENT</td>
<td>-450</td>
</tr>
<tr>
<td>A5/197</td>
<td>NET MOVEMENT</td>
<td>1760</td>
</tr>
<tr>
<td>B41/728</td>
<td>NET MOVEMENT</td>
<td>7</td>
</tr>
</tbody>
</table>

No attention need be paid to such refinements as skipping over the perforations at the end of each sheet of paper.'
step 1: structures
step 1: structures
step 1: structures

- **SMF**
  - **PART-GROUP**
    - **MOVEMT-REC**
      - **ISSUE**
      - **RECEIPT**
  - **REPORT**
    - **SUMMARY LINE**
step 1: structures
step 2: operations

1. smf = open (...)
2. close (smf)
3. rec = read (smf)
4. display (pno, net)
5. pno = rec.part_number
6. net = 0
7. net = net + rec.quantity
8. net = net - rec.quantity
1. smf = open (...) 
2. close (smf) 
3. rec = read (smf) 
4. display (pno, net) 
5. pno = rec.part_number 
6. net = 0 
7. net = net + rec.quantity 
8. net = net - rec.quantity
the final program

smf = open (...); rec = read (smf);

\textbf{while} (!eof (rec)) {
    pno = rec.part_number;
    net = 0;
    \textbf{while} (!eof (rec) && pno == rec.part_number) {
        if (rec.code == ISSUE) {
            net = net - rec.quantity;
        } else {
            net = net + rec.quantity;
        }
        rec = read (smf);
    }
    display (pno, net);
}

close (smf);
Hoare's approach
programs & traces
smf = open (...); rec = read (smf);

while (!eof (rec)) {
    pno = rec.part_number;
    net = 0;
    while (...) {
        if (rec.code == ISSUE)
            net = net - rec.quantity;
        else
            net = net + rec.quantity;
        rec = read (smf);
    }
    display (pno, net);
}

close (smf);
```python
smf = open (...); rec = read (smf);
while (!eof (rec)) {
    pno = rec.part_number;
    net = 0;
    while (...) {
        if (rec.code == ISSUE)
            net = net - rec.quantity;
        else
            net = net + rec.quantity;
        rec = read (smf);
    }
    display (pno, net);
}
close (smf);
```
smf = open (...); rec = read (smf);

while (!eof (rec)) {
    pno = rec.part_number;
    net = 0;
    while (...)
    {
        if (rec.code == ISSUE)
            net = net - rec.quantity;
        else
            net = net + rec.quantity;
        rec = read (smf);
    }
    display (pno, net);
}
close (smf);
smf = open (...); rec = read (smf);
while (!eof (rec)) {
    pno = rec.part_number;
    net = 0;
    while (...) {
        if (rec.code == ISSUE)
            net = net - rec.quantity;
        else
            net = net + rec.quantity;
        rec = read (smf);
    }
    display (pno, net);
}
close (smf);

a trace
<open, read, display, close>
smf = open (...); rec = read (smf);

while (!eof (rec)) {
    pno = rec.part_number;
    net = 0;
    while (...) {
        if (rec.code == ISSUE)
            net = net - rec.quantity;
        else
            net = net + rec.quantity;
        rec = read (smf);
    }
    display (pno, net);
}

close (smf);

a trace
<open, read, display, close>

selective trace on input
<open, read, close>
smf = open (...); rec = read (smf);

while (!eof (rec)) {
    pno = rec.part_number;
    net = 0;
    while (...) {
        if (rec.code == ISSUE)
            net = net - rec.quantity;
        else
            net = net + rec.quantity;
        rec = read (smf);
    }
    display (pno, net);
}

close (smf);

programs & traces

a trace
<open, read, display, close>

selective trace on input
<open, read, close>
smf = open (...); rec = read (smf);
while (!eof (rec)) {
    pno = rec.part_number;
    net = 0;
    while (...) {
        if (rec.code == ISSUE)
            net = net - rec.quantity;
        else
            net = net + rec.quantity;
        rec = read (smf);
    }
    display (pno, net);
}
close (smf);

a trace
<open, read, display, close>

selective trace on input
<open, read, close>

selective trace on output
<display>
selective programs & traces

```plaintext
smf = open (...); rec = read (smf);
while (!eof (rec)) {
    pno = rec.part_number;
    net = 0;
    while (...) {
        if (rec.code == ISSUE)
            net = net - rec.quantity;
        else
            net = net + rec.quantity;
        rec = read (smf);
    }
    display (pno, net);
}
close (smf);
```

a trace

```plaintext
<open, read, display, close>
```

selective trace on input

```plaintext
<open, read, close>
```

selective program

```plaintext
smf = open (...); rec = read (smf);
while (...) {
    while (...) {
        rec = read (smf);
    }
    display (pno, net);
}
close (smf);
```
merging selective programs
merging selective programs

Diagram:

- SMF
  - open
  - read
  - GROUPS
    - close
    - PART-GROUP
      - RECS
        - read
        - MOVEMT-REC
          - ISSUE
          - RECEIPT
merging selective programs

- open
- read
- GROUPS
- close
- PART-GROUP
- RECS
- read
- MOVEMENT-REC
- ISSUE
- RECEIPT
- REPORT
- BODY
- SUMMARY LINE
- display
merging selective programs
merging selective programs

- open
- read
- close

- GROUPS/BODY
- SMF/REPORT

- PART-GROUP/SUMMARY LINE
- RECS
- read
- display

- MOVEMENT-REC
- ISSUE
- RECEIPT
formalization

annotating with non-terminals
- left (right) annotation puts non-terminal symbol at start (end)
- now traces include non-terminals!

correspondences
- two symbols correspond if they alternate \(<a, b, a, b, ...>\)

transformation rules
- how to make structures match?
- apply algebraic rewrites, eg. \(Q = Q \cup Q\)
implications

methodical matching
\· merged structure correct by construction
\· in JSP, not formal but can check after

can project on variables too
\· operation allocation subsumed

structures clash relative
\· INPUT; OUTPUT always possible
\· can do this at any level

paper available online at http://tinyurl.com/hoarejsp