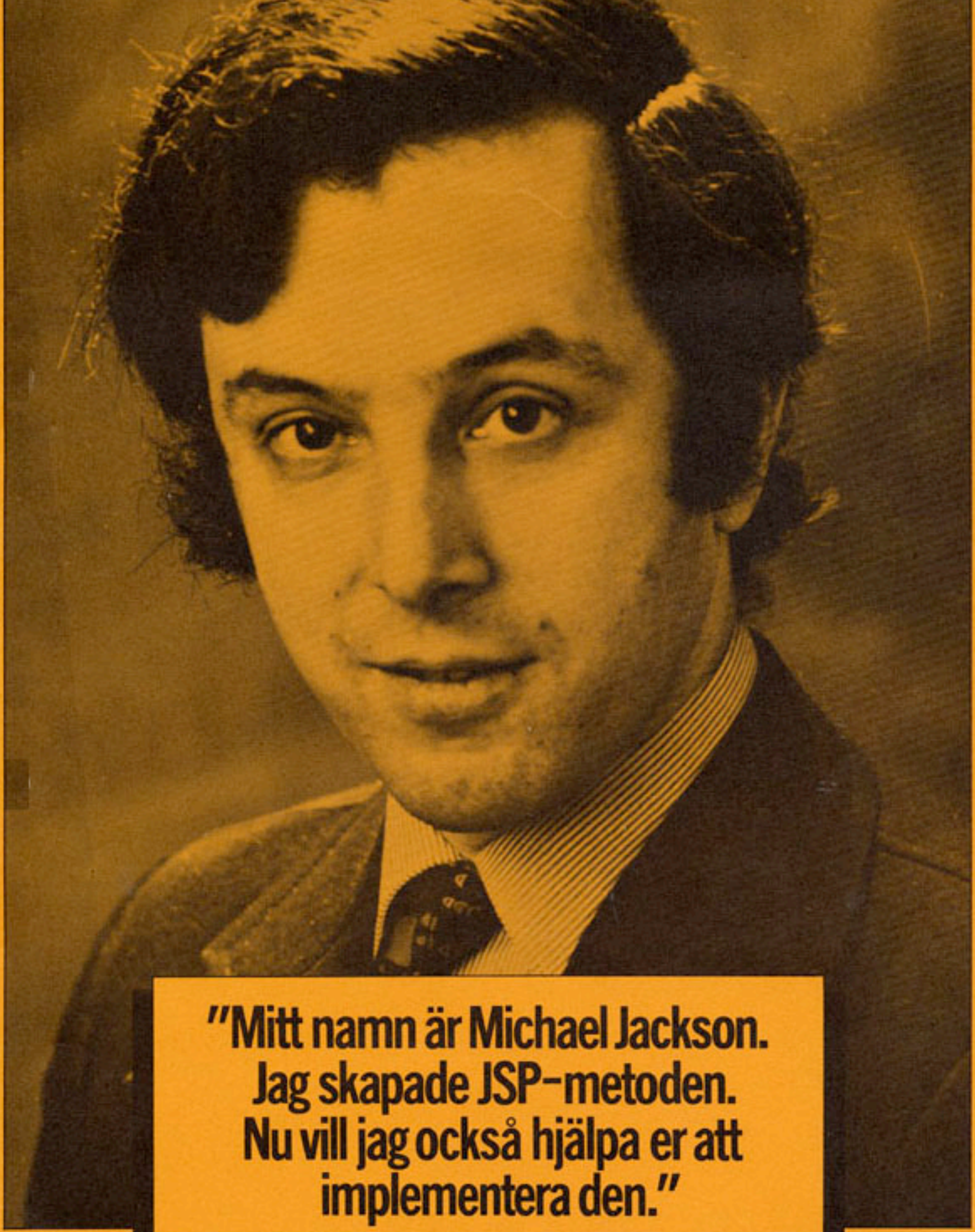
Hoare on JSP

A report on JSP by C.A.R. Hoare, 1977

A Tribute to Michael Jackson · Vancouver · May 2009

Daniel Jackson · MIT



"Mitt namn är Michael Jackson. Jag skapade JSP-metoden. Nu vill jag också hjälpa er att implementera den."

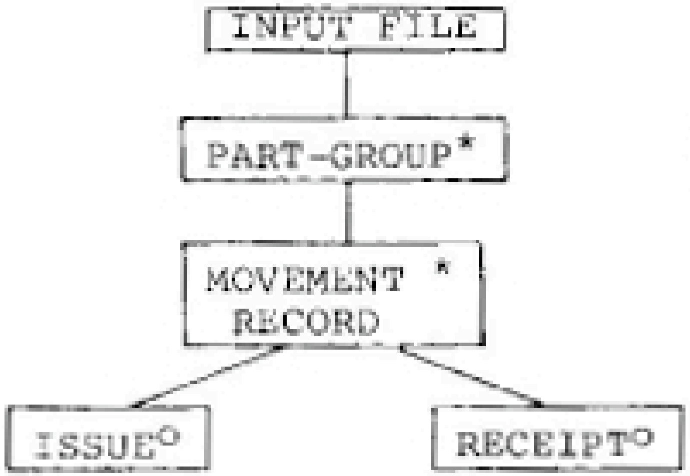
'The stores section in a factory issues and receives parts. Each issue and each receipt is recorded on a punched card: the card contains the part-number, the movement type (I for issue, R for receipt) and the quantity. The cards have already been copied to magnetic tape and sorted into part-number order. The program to be written will produce a simple summary of the net movement of each part. The format of the summary is:

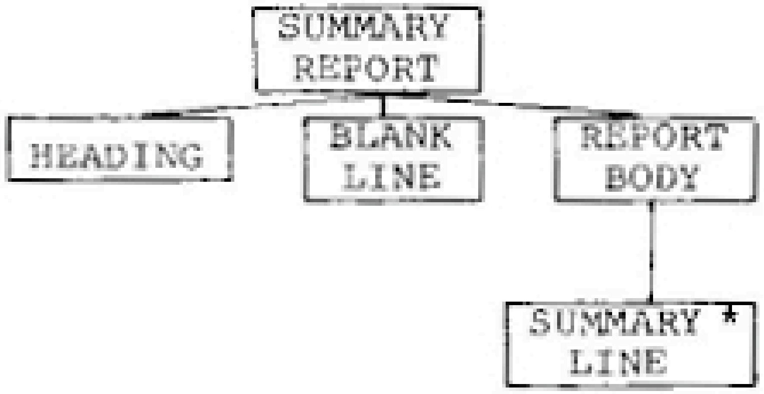
```
STORES MOVEMENTS SUMMARY

A5/132       NET MOVEMENT        -450
A5/197       NET MOVEMENT        1760
B41/728      NET MOVEMENT           7
                  .
                  .
                  .
```

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,*

The Michael Jackson Design Technique: A study of the theory with applications

C.A.R.Hoare

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.

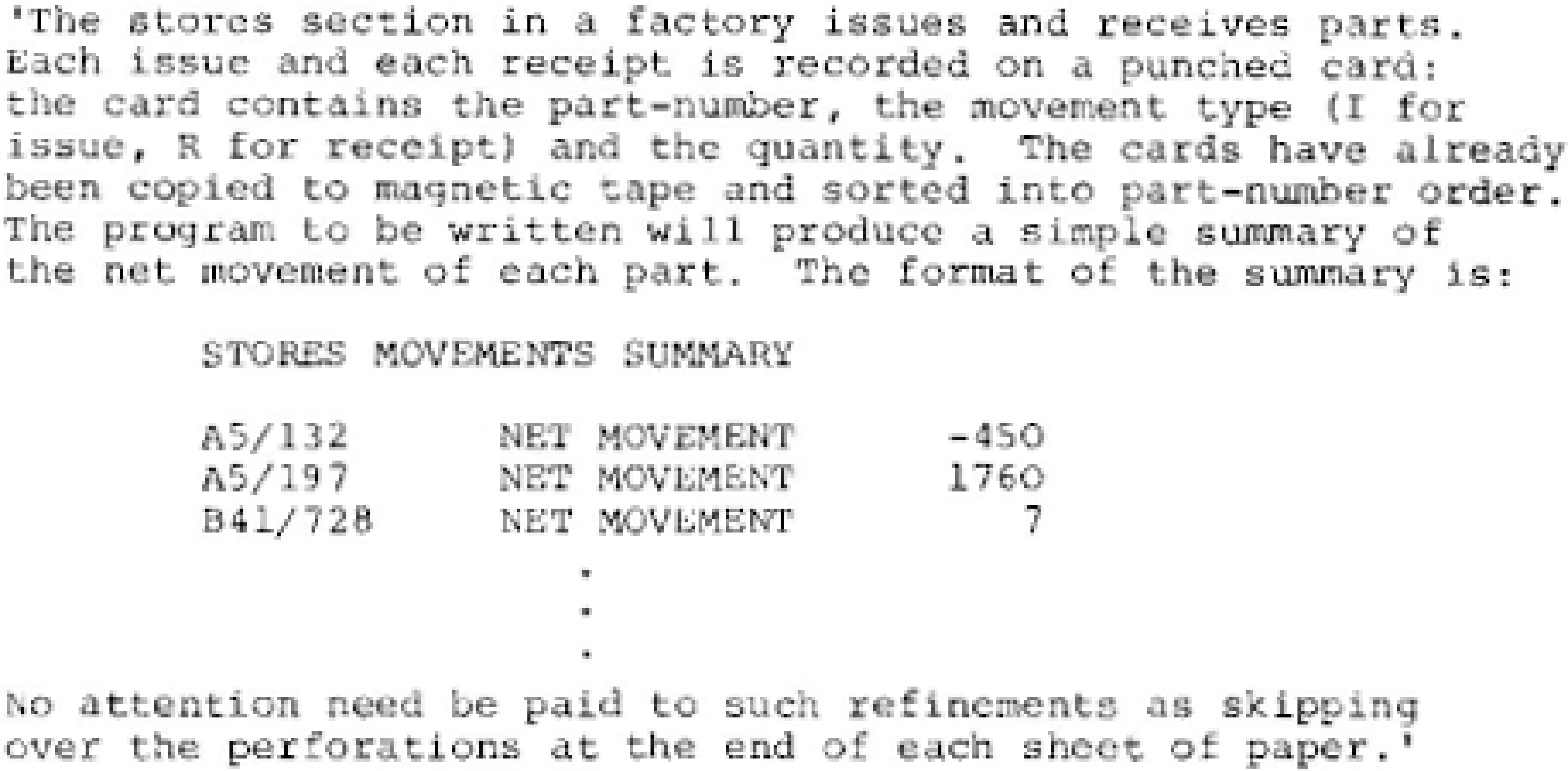
(b) For a sequence of commands (say `P ; Q`), every possible trace consists of a trace of `P` followed by a trace of `Q` (and similarly for sequences longer than two).

(c) For a selection (say `P ∪ 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

a JSP example

stores movement problem



'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:

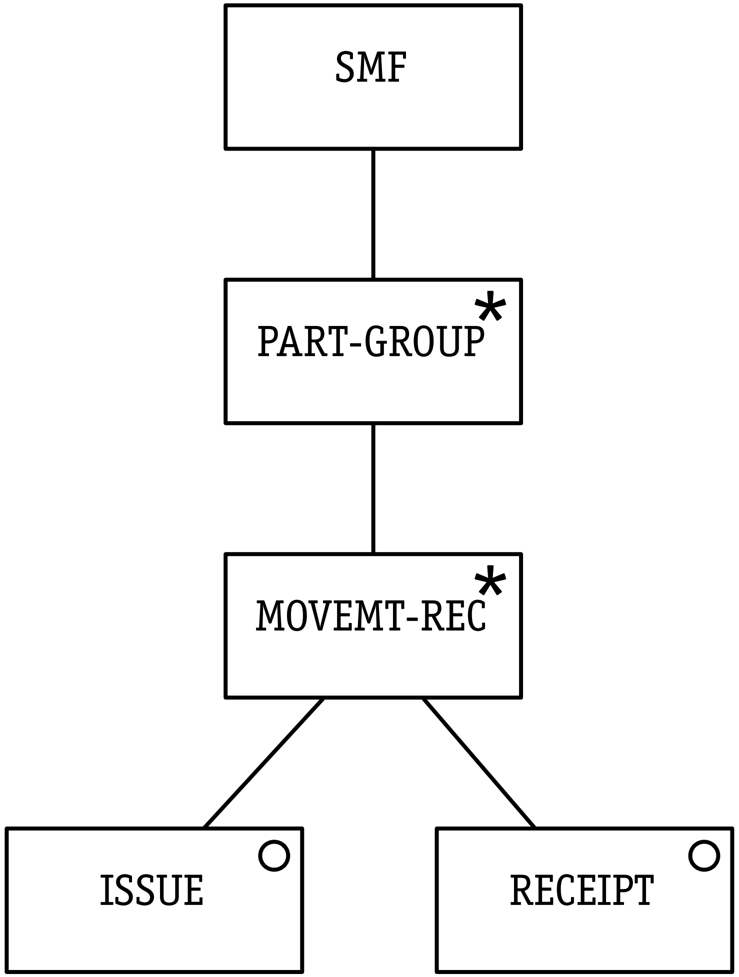
```
        STORES MOVEMENTS SUMMARY

        A5/132       NET MOVEMENT       -450
        A5/197       NET MOVEMENT       1760
        B41/728      NET MOVEMENT          7
                         .
                         .
                         .
```

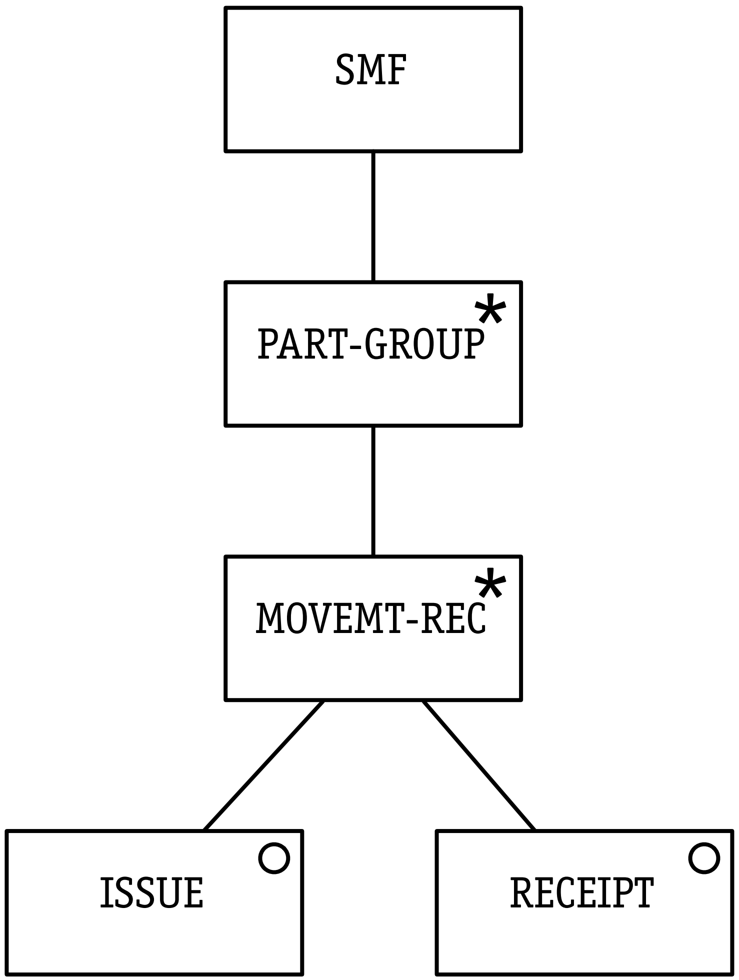
No attention need be paid to such refinements as skipping over the perforations at the end of each sheet of paper.'

step 1: structures

step 1: structures



step 1: structures



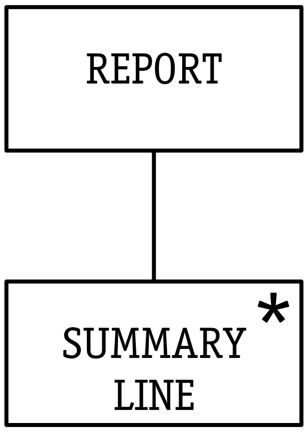
SMF

PART-GROUP *

MOVEMT-REC *

ISSUE ○

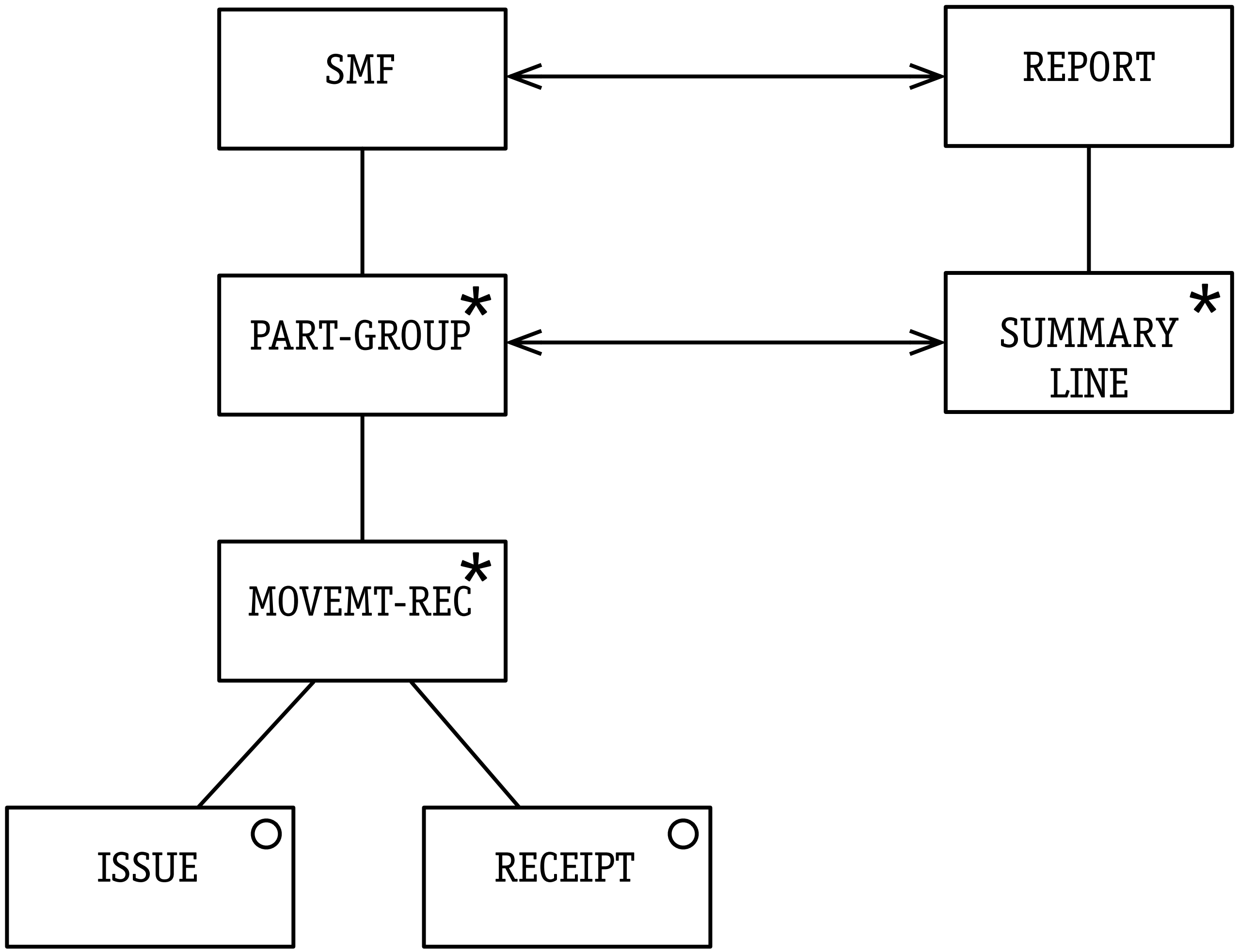
RECEIPT ○



REPORT

SUMMARY LINE *

step 1: structures

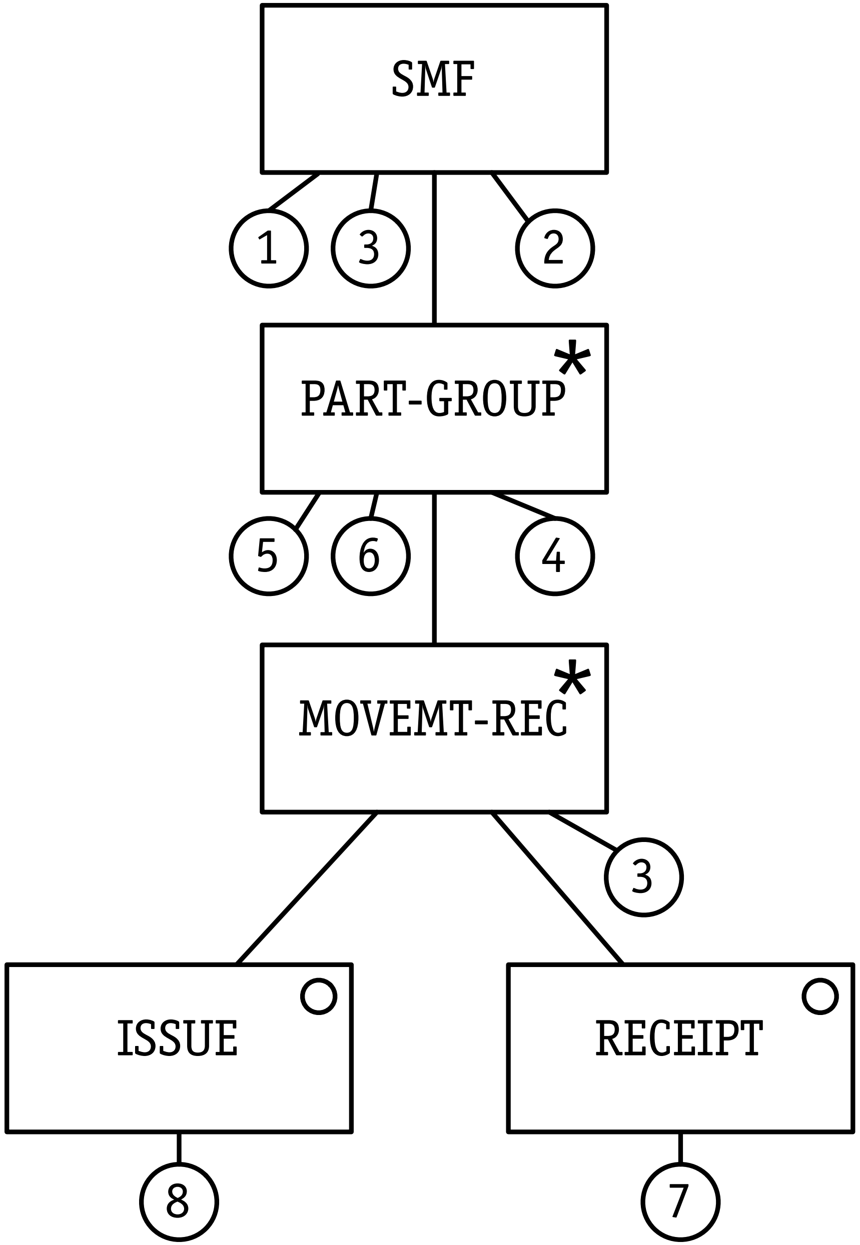


step 2: operations

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

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



the final program

```
smf = open (...); rec = read (smf);
while (!eof (rec)) {
    pno = rec.part_number;
    net = 0;
    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

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);
```

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);
```

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);
```

a trace

<open, read, display, close>

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);
```

a trace

<open, read, display, close>

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);
```

a trace

<open, read, display, close>

selective trace on input

<open, read, close>

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);
```

a trace

<open, read, display, close>

selective trace on input

<open, read, close>

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);
```

a trace

<open, read, display, close>

selective trace on input

<open, read, close>

selective trace on output

<display>

selective 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);
```

a trace

<open, read, display, close>

selective trace on input

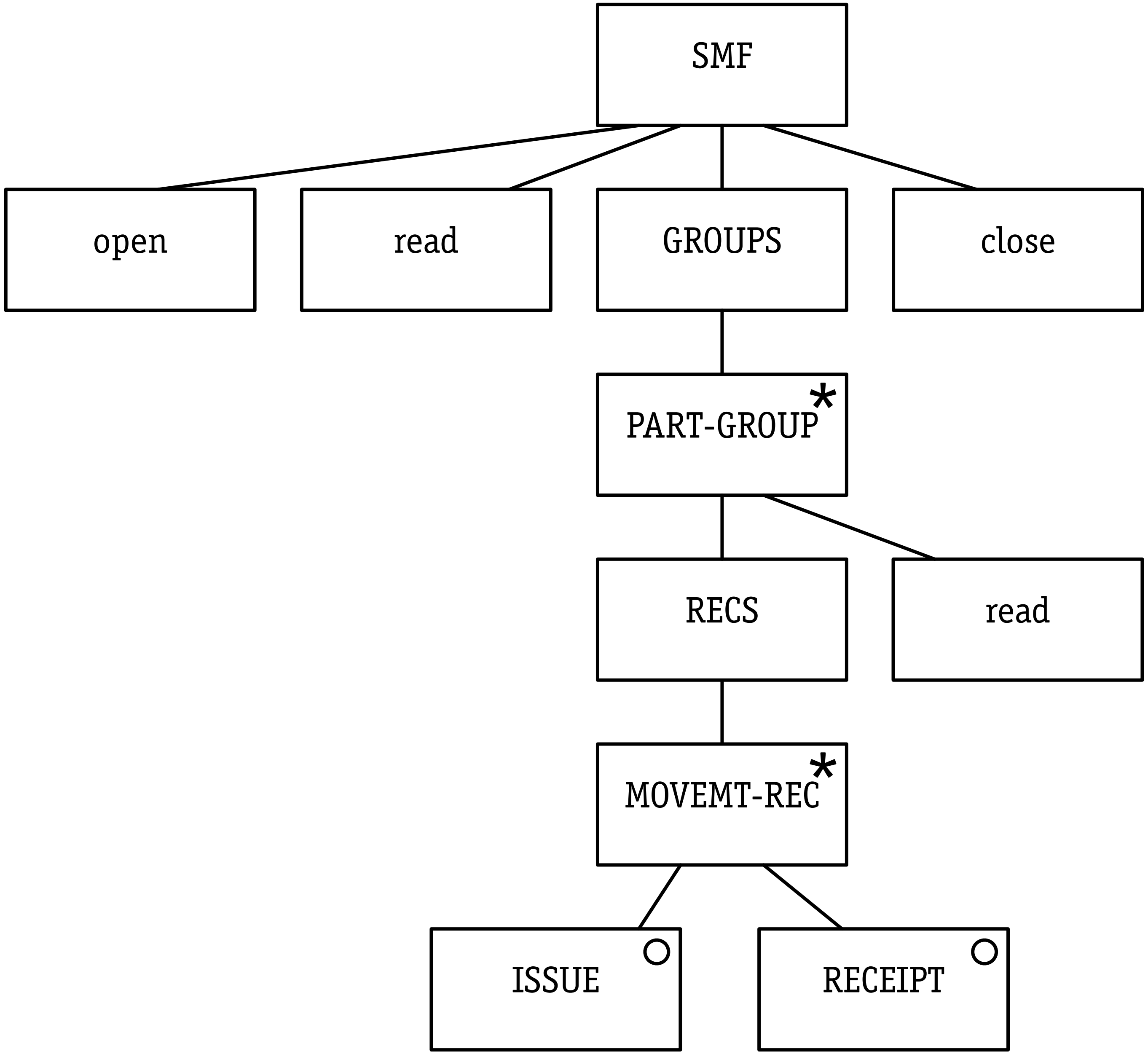
<open, read, close>

selective program

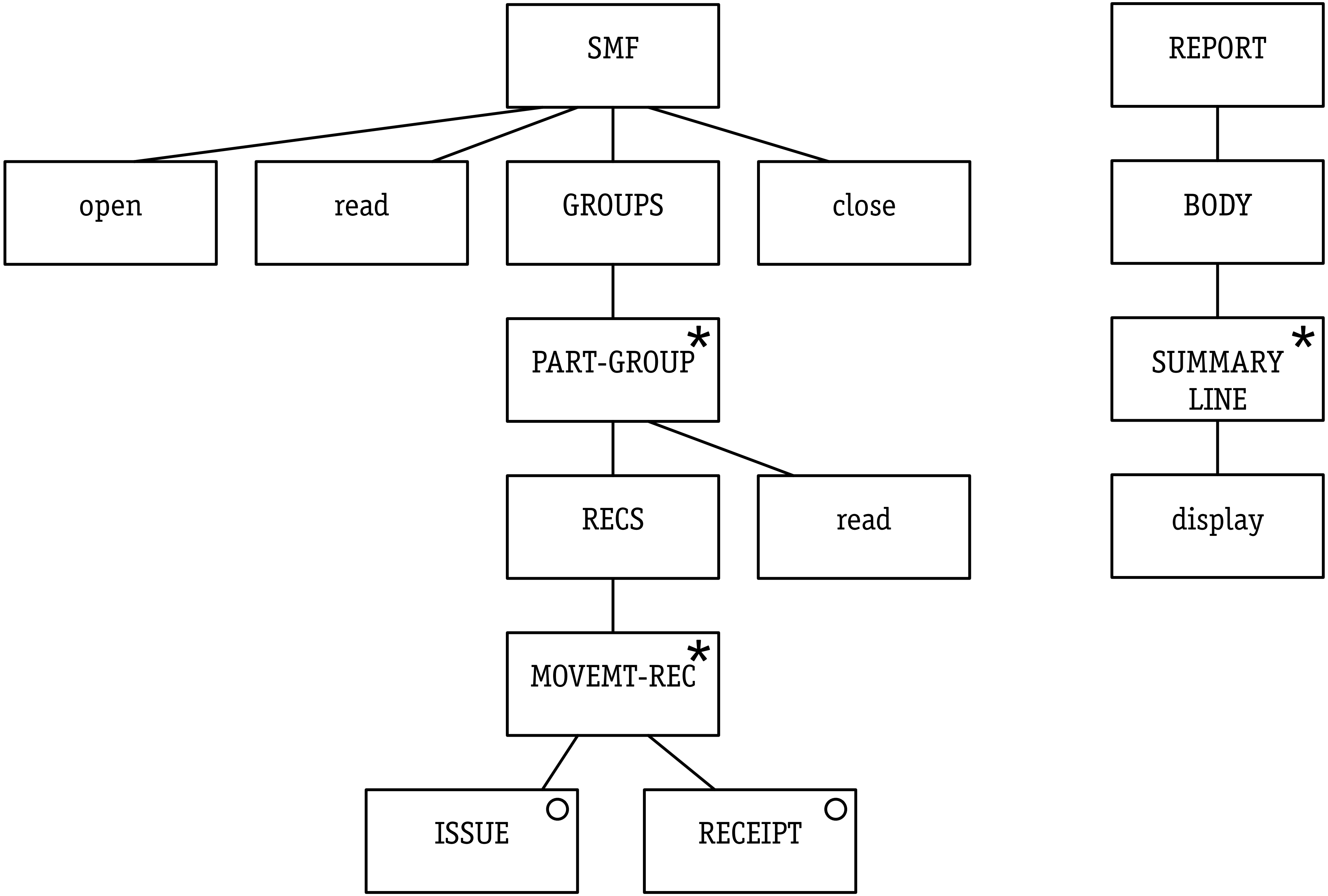
```
smf = open (...); rec = read (smf);
while (...)  {
   while (...) {
      rec = read (smf);
      }
   }
```

merging selective programs

merging selective programs



merging selective programs



SMF

open

read

GROUPS

close

PART-GROUP *

RECS

read

MOVEMT-REC *

ISSUE o

RECEIPT o

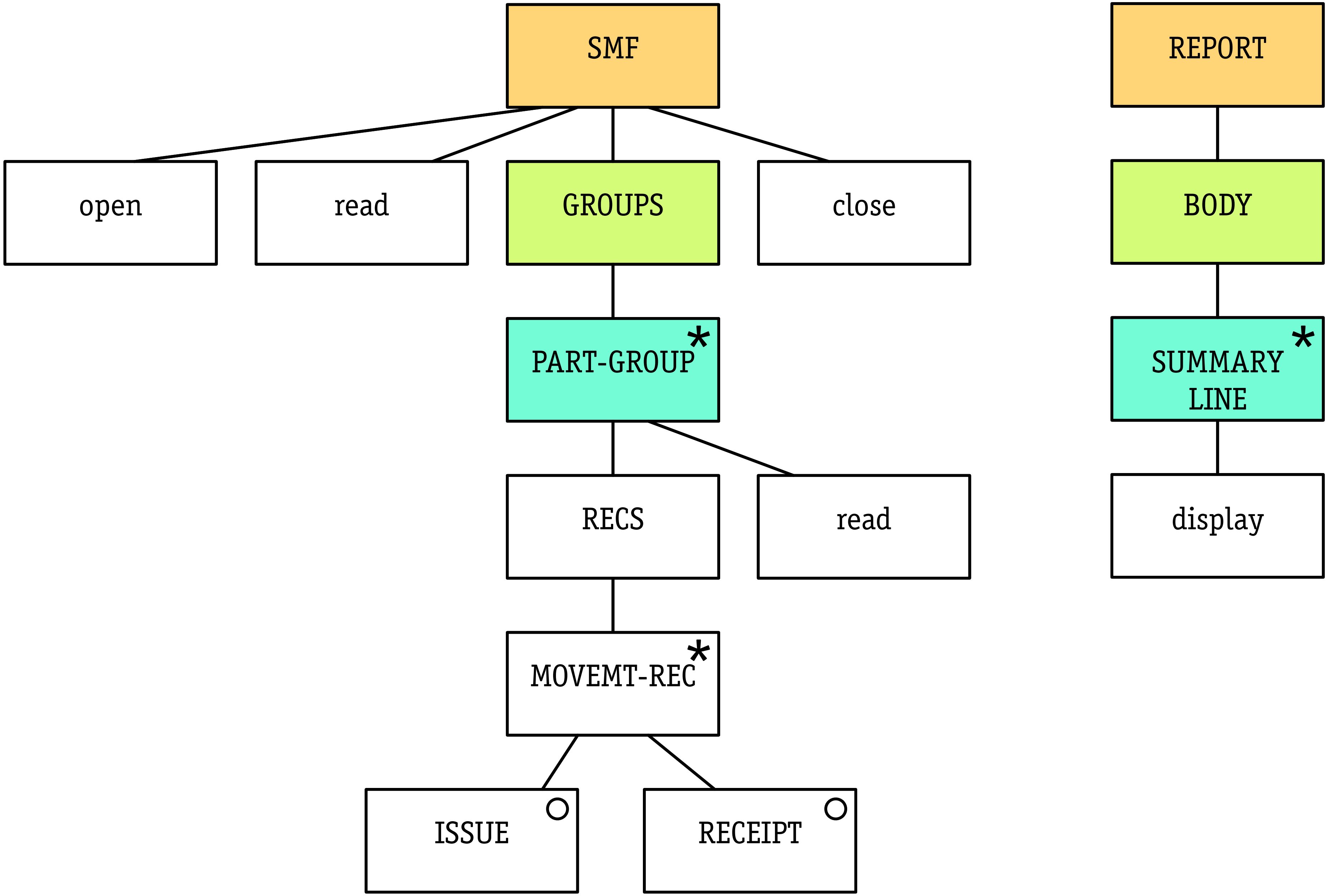
REPORT

BODY

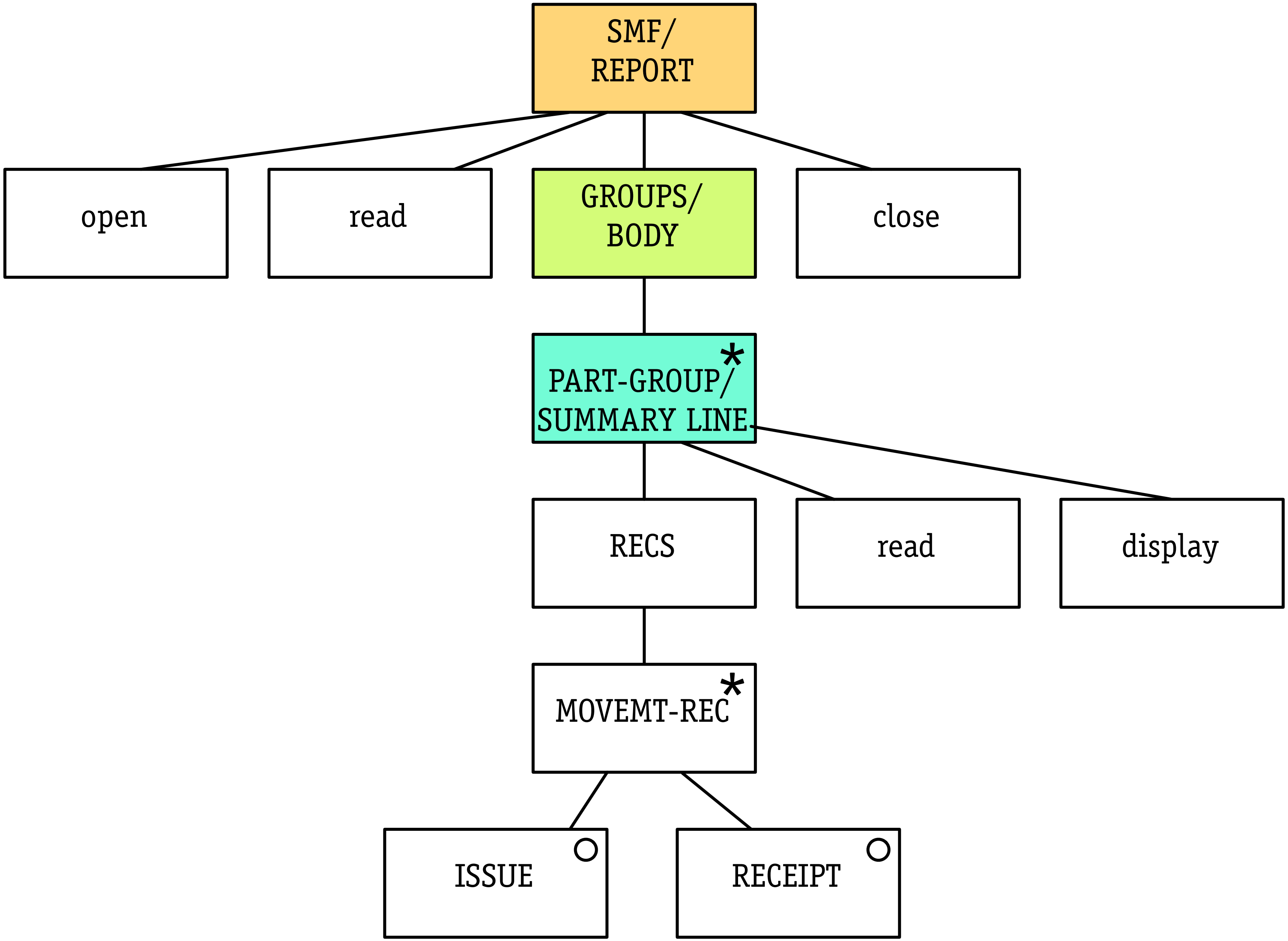
SUMMARY LINE *

display

merging selective programs



merging selective programs



SMF/ REPORT

open

read

GROUPS/ BODY

close

PART-GROUP/ SUMMARY LINE *

RECS

read

display

MOVEMT-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