David L. Parnas
On a 'Buzzword': Hierarchical Structure

Information Processing 74, IFIP Congress 74, North Holland,
1974
pp. 336–339

ON A 'BUZZWORD': HIERARCHICAL STRUCTURE

David PARNAS

Technische Hochschule Darmstadt, Fachbereich Informatik
Research Group on Operating Systems I
Steubenplatz 12, 61 Darmstadt, West Germany

This paper discusses the use of the term "hierarchically structured" to describe the design of operating systems. Although the various uses of this term are often considered to be closely related, close examination of the use of the term shows that it has a number of quite different meanings. For example, one can find two different senses of "hierarchy" in a single operating system [3] and [6]. An understanding of the different meanings of the term is essential, if a designer wishes to apply recent work in Software Engineering and Design Methodology. This paper attempts to provide such an understanding.

INTRODUCTION

The phrase "hierarchical structure" has become a buzzword in the computer field. For many it has acquired a connotation so positive that it is akin to the quality of being a good mother. Others have rejected it as being an unrealistic restriction on the system [1]. This paper attempts to give some meaning to the term by reviewing some of the ways that the term has been used in various operating systems (e.g. T.H.E. [3], MULTICS [12], and the RC4ooo [8]) and providing some better definitions. Uses of the term, which had been considered equivalent or closely related, are shown to be independant. Discussions of the advantages and disadvantages of the various hierarchical restrictions are included.

GENERAL PROPERTIES OF ALL USES OF THE PHRASE "HIERARCHICAL STRUCTURE"

As discussed earlier [2], the word "structure" refers to a partial description of a system showing it as a collection of parts and showing some relations between the parts. We can term such a structure <u>hierarchical</u>, if a relation or predicate on pairs of the parts ($R(\alpha,\beta)$) allows us to define levels by saying that

1. Level 0 is the set of parts α such that there does not exist a β such that $R(\alpha,\beta)$, and
2. Level i is the set of parts α such that
   a) there exists a β on level i-1 such that $R(\alpha,\beta)$ and
   b) if $R(\alpha,\gamma)$ then γ is on level i-1 <u>or lower</u>.

This is possible with a relation R only if the directed graph representing R has no loops.

The above definition is the most precise reasonably simple definition, which encompasses all uses of the word in the computer literature. This suggests that the statement "our Operating System has a hierarchical structure" carries no information at all. <u>Any</u> system can be represented as a hierarchical system with one level and one part; more importantly, it is possible to divide <u>any</u> system into parts and contrive a relation such that the system has a hierarchical structure. Before such a statement can carry any information at all, the way that the system is divided into parts and the nature of the relation must be specified.

The decision to produce a hierarchically structured system may restrict the class of possible systems, and may, therefore, introduce disadvantages as well as the desired advantages. In the remainder of this paper we shall introduce a variety of definitions for "hierarchical structure", and mention some advantages and disadvantages of the restriction imposed by these definitions.

1. THE PROGRAM HIERARCHY

Prof. E.W. Dijkstra in his paper on the T.H.E. system and in later papers on structured programming [3] and [4] has demonstrated the value of programming using layers of abstract machines. We venture the following definition for this program hierarchy. The parts of the system are subprograms, which may be called as if they were procedures.* We assume that

* They may be expanded as MACROS.

each such program has a specified purpose (e.g.
FNO :: = find next odd number in sequence or invoke
DONE if there is none). The relation "uses" may be
defined by $USES(p_i, p_j) = $ iff p_i calls p_j and p_i will
be considered incorrect if p_j does not function
properly.

With the last clause we intend to imply that, our
example, FNO does not "use" DONE in the sense de-
fined here. The task of FNO is to invoke DONE; the
purpose and "correctness" of DONE is irrelevant to
FNO. Without excepting such calls, we could not
consider a program to be higher in the hierarchy
than the machine, which it uses. Most machines have
"trap" facilities, and invoke software routines,
when trap conditions occur.

A program divided into a set of subprograms may be
said to be hierarchically structured, when the re-
lation "uses" defines levels as described above.
The term "abstract machine" is commonly used,
because the relation between the lower level pro-
grams and the higher level programs is analoguous
to the relation between hardware and software.

A few remarks are necessary here. First, we do not
claim that the only good programs are hierarchically
structured programs. Second, we point out that the
way that the program is divided into subprograms can
be rather arbitrary. For any program, some decom-
positions into subprograms may reveal a hierarchi-
cal structure, while other decompositions may show
a graph with loops in it. As demonstrated in the
simple example above, the specification of each
program's purpose is critical!

The purpose of the restriction on program structure
implied by this definition, is twofold. First, the
calling program should be able to ignore the inter-
nal workings of the called program; the called pro-
gram should make no assumptions about the internal
structure of the calling program. Allowing the
called program to call its user, might make this
more difficult since each would have to be designed

to work properly in the situations where it could be called by the other.

The second purpose might be termed "ease of subsetting". When a program has this "program hierarchy", the lower levels may always be used without the higher levels, when the higher levels are not ready or their services are not needed. An example of non-hierarchical systems would be one in which the "lower level" scheduling programs made use of the "high level" file system for storage of information about the tasks that it schedules. Assuming that nothing useful could be done without the scheduler, no subset of the system that did not include the file system could exist. The file system (usually a complex and "buggy" piece of software) could not be developped using the remainder of the system as a "virtual machine".

For those who argue that the hierarchical structuring proposed in this section prevents the use of recursive programming techniques, we remind them of the freedom available in choosing a decomposition into subprograms. If there exists a subset of the programs, which call each other recursively, we can view the group as a single program for this analysis and then consider the remaining structure to see, if it is hierarchical. In looking for possible subsets of a system, we must either include or exclude this group of programs as a single program.

One more remark: please, note that the division of the program into levels by the above discussed relation has no necessary connection with the division of the programs into modules as discussed in [5]. This is discussed further later (section 6).

2. THE "HABERMANN" HIERARCHY IN THE T.H.E. SYSTEM

The T.H.E. system was also hierarchical in another sense. In order to make the system relatively insensitive to the number of processors and their relative speeds, the system was designed as a set of "parallel sequential processes". The activities in

the system were organized into "processes" such
that the sequence of events within a process was
relatively easy to predict, but the sequencing of
events in different processes were considered un-
predictable (the relative speeds of the processes
were considered unknown). Resource allocation was
done in terms of the processes and the processes
exchanged work assignments and information. In
carrying out a task, a process could assign part of
the task to another process in the system.

One important relation between the processes in
such a system is the relation "gives work to". In
his thesis [6] Habermann assumed that "gives work
to" defined a hierarchy to prove "harmonious coop-
eration". If we have an Operating System we want to
show that a request of the system will generate on-
ly a finite (and reasonably small) number of re-
quests to individual processes before the original
request is satisfied. If the relation "gives work
to" defines a hierarchy, we can prove our result by
examining each process seperately to make sure that
every request to it results in only a finite number
of requests to other processes. If the relation is
not hierarchical, a more difficult, "global", anal-
ysis would be required.

Restricting "gives work to" so that it defines a
hierarchy helps in the establishment of the "well-
behavedness", but it is certainly not a necessary
condition for "harmonious cooperation".*

*This restriction is also valuable in human organiza-
tions. Where requests for administrative work flow
only in one direction things go relatively smoothly,
but in departments where the "leader" constantly re-
fers requests "downward" to committees (which can
themselves send requests to the "leader") we often
find the system filling up with uncompleted tasks
and a correspondingly large increase in overhead.

In the T.H.E. system the two hierarchies described
above coincided. Every level of abstraction was
achieved by the introduction of parallel processes
and these processes only gave work to those
written to implement lower levels in the program
hierarchy. One should not draw general conclusions
about system structure on the basis of this coinci-
dence. For example, the remark that "building a
system with more levels than were found in the T.H.E.
system is undesirable, because it introduces more
queues" is often heard because of this coincidence.
The later work by Dijkstra on structured programming
[21] shows that the levels of abstraction are useful
when there is only one process. Further, the "Haber-
mann hierarchy" is useful, when the processes are
controlled by badly structured programs. Adding
levels in the program hierarchy need not introduce
new processes or queues. Adding processes can be done
without writing new programs.

The "program hierarchy" is only significant at times
when humans are working with the program (e.g. when
the program is being constructed or changed). If the
programs were all implemented as macros, there would
be no trace of this hierarchy in the running system.
The "Habermann hierarchy" is a restriction on the
run time behavior of the system. The theorems proven
by Habermann would hold even if a process that is
controlled by a program written at a low level in
the program hierarchy "gave work to" a process which
was controlled by a program originally written at a
higher level in the program hierarchy. There are
also no detrimental effects on the program hierarchy
provided that the programs written at the lower
level are not written in terms of programs at the
higher level. Readers are referred to "Flatland" [7].

3. HIERARCHICAL STRUCTURES RELATING TO RESOURCE

OWNERSHIP AND ALLOCATION

The RC4ooo system [8] and [9] enforced a hierarchi-
cal relation based upon the ownership of memory. A
generalization of that hierarchical structure has

been proposed by Varney [lo] and similar hierarchi-
cal relationships are to be found in various com-
mercial operating systems, though they are not often
formally described.

In the RC4ooo system the objects were processes and
the relation was "allocated a memory region to".
Varney proposes extending the relation so that the
hierarchical structure controlled the allocation of
other resources as well. (In the RC4ooo systems
specific areas of memory were allocated, but that
was primarily a result of the lack of virtual memory
hardware; in most systems of interest now, we can
allocate quantities of a resource without allocating
the specific physical resources until they are
actually used). In many commercial systems we also
find that resources are not allocated directly to
the processes which use them. They are allocated to
administrative units, who, in turn, may allocate
them to other processes. In these systems we do not
find any loops in the graph of "allocates resources
to", and the relation defines a hierarchy, which is
closely related to the RC4ooo structure.

This relation was not a significant one in the
T.H.E. system, where allocating was done by a cen-
tral allocator called a BANKER. Again this sense of
hierarchy is not strongly related to the others, and
if it is present with one or more of the others,
they need not coincide.

The disadvantage of a non-trivial hierarchy (the
hierarchy is present in a trivial form even in the
T.H.E. system) of this sort are (1) poor resource
utilization that may occur when some processes in the
system are short of resources while other processes,
under a different allocator in the hierarchy, have an
excess; (2) high overhead that occurs when resources
are tight. Requests for more resources must always
go up all the levels of the hierarchy before being
denied or granted. The central "banker" does not have
these disadvantages. A central resource allocator,
however, becomes complicated in situations where
groups of related processes wish to dynamically share

resources without influence by other such groups.
Such situations can arise in systems that are used in
real time by independant groups of users. The T.H.E.
system did not have such problems and as a result,
centralized resource allocation was quite natural.

It is this particular hierarchical relation which
the Hydra group rejected. They did not mean to reject
the general notion of hierarchical structure as
suggested in the original report [1] and [11].

4. PROTECTION HIERARCHIES A LA MULTICS

Still another hierarchy can be found in the MULTICS
system. The conventional two level approach to oper-
ating systems (low level called the supervisor, next
level the users) has been generalized to a sequence
of levels in the supervisor called "rings". The set
of programs within a MULTICS process is organized in
a hierarchical structure, the lower levels being
known as the inner rings, and the higher levels being
known as outer rings. Although the objects are pro-
grams, this relation is not the program hierarchy
discussed in section 1. Calls occur in both direc-
tions and lower level programs may use higher level
ones to get their work done [12].

Noting that certain data are much more crucial to
operation of the system than other data, and that
certain procedures are much more critical to the
overall operation of the system than others, the
designers have used this as the basis of their hier-
archy. The data to which the system is most sensitive
are controlled by the inner ring procedures, and
transfers to those programs are very carefully con-
trolled. Inner ring procedures have unrestricted
access to programs and data in the outer rings. The
outer rings contain data and procedures that effect
a relatively small number of users and hence are
less "sensitive". The hierarchy is most easily de-
fined in terms of a relation "can be accessed by"
since "sensitivity" in the sense used above is diffi-
cult to define. Low levels have unrestricted access
to higher levels, but not vice versa.

It is clear that placing restrictions on the rela-
tion "can be accessed by" is important to system
reliability and security.

It has, however, been suggested that by insisting
that the relation "can be accessed by" be a hierar-
chy, we prevent certain accessibility patterns that
might be desired. We might have three segments in
which A requires access to B, B to C, and C to A.
No other access rights are needed or desirable. If
we insist that "can be accessed by" define a hierar-
chy, we must (in this case) use the trivial hierar-
chy in which A, B, C are considered one part.

In the view of the author, the member of pairs in
the relation "can be accessed by" should be mini-
mized, but he sees no advantage in insisting that it
define a hierarchy [13] and [14].

The actual MULTICS restriction is even stronger than
requiring a hierarchy. Within a process, the rela-
tion must be a complete ordering.

5. HIERARCHIES AND "TOP DOWN" DESIGN METHODOLOGY

About the time that the T.H.E. system work appeared,
it became popular to discuss design methods using
such terms as "top down" and "outside in"[15], [16],
and [17]. The simultaneous appearance of papers
suggesting how to design well and a well designed
system led to the unfounded assumption that the
T.H.E. system had been the result of a "top down"
design process. Even in more recent work [18] top
down design and structured programming are consid-
ered almost synonymous.

Actually "outside in" was a much better term for
what was intended, than was "top down"! The inten-
tion was to begin with a description of the system's
user interface, and work in small, verifiable steps
towards the implementation. The "top" in that hier-
archy consisted of those parts of the system that
were visible to the user. In a system designed ac-
cording to the "program hierarchy", the lower level

functions will be used by the higher level functions,
but some of them may also be visible to the user
(store and load, for example). Some functions on
higher levels may not be available to him (Restart
system). Those participants in the design of the
T.H.E. system with whom I have discussed the ques-
tion [19], report that they did not proceed with the
design of the higher levels first.

6. HIERARCHICAL STRUCTURE AND DECOMPOSITION

INTO MODULES

Often one wants to view a system as divided into
"modules" (e.g. with the purpose outlined in [5] and
[2o]). This division defines a relation "part of".
A group of sub-programs is collected into a module,
groups of modules collected into bigger modules, etc.
This process defines a relation "part of" whose
graph is clearly loop-free. It remains loop-free
even if we allow programs or modules to be part of
several modules - the part never includes the whole.

Note that we may allow programs in one module to
call programs in another module, so that the module
hierarchy just defined need not have any connection
with the program hierarchy. Even allowing recursive
calls between modules does not defeat the purpose of
the modular decomposition (e.g. flexibility) [5],
provided that programs in one module do not assume
much about the programs in another.

7. LEVELS OF LANGUAGE

It is so common to hear phrases such as "high level
language", "low level language" and "linguistic
level" that it is necessary to comment on the re-
lation between the implied language hierarchy and
the hierarchies discussed in the earlier sections
of this paper. It would be nice, if, for example,
the higher level languages were the languages of the
higher level "abstract machines" in the program
hierarchy. Unfortunately, this author can find no
such relation and cannot define the hierarchy

that is implied in the use of those phrases. In moments of scepticism one might suggest that the relation is "less efficient than" or "has a bigger grammar than" or "has a bigger compiler than", however, none of those phrases suggests an ordering, which is completely consistant with the use of the term. It would be nice, if the next person to use the phrase "higher level language" in a paper would define the hierarchy to which he refers.

SUMMARY

The computer system design literature now contains quite a number of valuable suggestions for improving the comprehensibility and predictability of computer systems by imposing a hierarchical structure on the programs. This paper has tried to demonstrate that, although these suggestions have been described in quite similar terms, the structures implied by those suggestions are not necessarily closely related. Each of the suggestions must be understood and evaluated (for its applicability to a particular system design problem) independantly. Further, we have tried to show that, while each of the suggestions offers some advantages over an "unstructured" design, there are also disadvantages, which must be considered. The main purpose of this paper has been to provide some guidance for those reading earlier literature and to suggest a way for future authors to include more precise definitions in their papers on design methods.

ACKNOWLEDGMENT

REFERENCES

[1] Wulf, Cohen, Coowin, Jones, Levin, Pierson, Pollach, Hydra: The Kernel of a Multiprogramming System, Technical Report, Computer Science Department, Carnegie-Mellon University.

[2] David L. Parnas, Information Distribution Aspects of Design Methodology, Proceedings of the 1971 IFIP Congress, Booklet TA/3, 26-3o.

[3] E.W. Dijkstra, The Structure of the T.H.E. Multiprogramming System, Communications of the ACM, vol 11, no. 5, May 1968, 341-346.

[4] E.W. Dijkstra, Complexity controlled by Hierarchical Ordering of Function and Variability, Software Engineering, NATO.

[5] David L. Parnas, On the Criteria to be Used in Decomposing Systems into Modules, Communications of the ACM, vol. 15, no. 12, December 1972, 1o53-1o58.

[6] A.N. Habermann, On the Harmonious Cooperation of Abstract Machines, Doctoral Dissertation, Technische Hogeschool Eindhoven, The Netherlands.

[7] Edwin A. Abbott, Flatland, the Romance of Many Dimensions, Dover Publications, Inc., New York, 1952.

[8] Per Brinch Hansen, The Nucleus of a Multiprogramming System, Communications of the ACM, vol. 13, no. 4, April 197o, 238-25o.

[9] RC4000 Reference Manuals for the Operating System, Regnecentralen Denmark.

[1o] R.C. Varney, Process Selection in a Hierarchical Operating System, SIGOPS Operating Review, June 1972.

[11] W. Wulf, C. Pierson, Private Discussions.

[12] R.W. Graham, Protection in an Information Processing Utility, Communications of the ACM, May 1968.

[13] W.R. Price and David L. Parnas, The Design of the Virtual Memory Aspects of a Virtual Machine, Proceedings of the SIGARCH-SIGOPS Workshop on Virtual Machines, March 1973.

[14] W.R. Price, Doctoral Dissertation, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., U.S.A.

[15] David L. Parnas and Darringer, SODAS and Methodology for System Design,Proceedings of 1967 FJCC.

[16] David L. Parnas, More on Design Methodology and Simulation, Proceedings of the 1969 SJCC.

[17] Zurcher and Randell, Iterative Multi-Level Modeling, Proceedings of the 1968 IFIP Congress.

[18] F.T. Baker, System Quality through Structured Programming, Proceedings of the 1972 FJCC.

[19] E.W. Dijkstra, A.N. Habermann, Private Discussions

[2o] David L. Parnas, Some Conclusions from an Experiment in Software Engineering, Proceedings of the 1972 FJCC.

[21] E.W. Dijkstra, A Short Introduction to the Art of Programming, in O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, Structured Programming, Academic Press, New York, 1972.