INFORMATION DISTRIBUTION ASPECTS OF DESIGN METHODOLOGY

by

D. L. Parnas

Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pennsylvania

February, 1971

This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (F44620-70-C-0107) and is monitored by the Air Force Office of Scientific Research.
This document has been approved for public release and sale; its distribution is unlimited.

ACKNOWLEDGEMENT

I am grateful to A. Perlis, H. Wactlar, and G. Bell for their suggestions after an early reading of this paper. I am deeply grateful to NV Philips-Electrologica, Apeldoorn, the Netherlands, for having provided me with the opportunity to study the problems of systems development in practice and by means of a direct involvement rather than a remote study. Although the problems discussed in this paper are apparently shared by everyone in the industry, the steps taken at Philips to improve the situation have provided me with valuable insight. Thanks are due to countless personnel, both at Philips and at several other institutions, who have been patient during my probing.

INFORMATION DISTRIBUTION ASPECTS OF DESIGN METHODOLOGY

D. L. Parnas
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania

ABSTRACT

The role of documentation in the design and implementation of complex systems is explored, resulting in suggestions in sharp contrast with current practice. The concept of system structure is studied by examining the meaning of the phrase "connections between modules". It is shown that several system design goals (each suggesting a partial time ordering of the decisions) may be inconsistent. Some properties of programmers are discussed. System documentation, which makes all information accessible to anyone working on the project, is discussed. The thesis that such information "broadcasting" is harmful, that it is helpful if most system information is hidden from most programmers, is supported by use of the above mentioned considerations as well as by examples. An information hiding technique of documentation is exhibited in the appendix.

IFIP CLASSIFICATION: 3

Language of Oral Presentation: English

Statement of Originality: In the opinion of the author the paper contains a number of conclusions which have not been discussed or published elsewhere. No paper similar in scope to this paper is being presented for publication elsewhere.

INFORMATION DISTRIBUTION ASPECTS OF DESIGN METHODOLOGY

D. L. Parnas
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania

INTRODUCTION

Papers on design methodology assume (1) that the methods used in system design affect strongly the quality of the final product; and (2) by selecting an appropriate methodology we can avoid many of the problems previously encountered in constructing large systems.

Under the heading "Design Methodology" a number of separate topics can be distinguished:

1. The order in which design decisions are made [1, 2, 3, 6]
2. The characteristics of the final product (e.g., what constitutes "good structure" for a system) [4, 5, 6, 7]
3. Methods of detecting errors in design decisions shortly after they are made [1, 2, 3, 5, 8, 9]
4. Specification techniques [12, 13]
5. Tools for system designers [1, 2, 3, 10, 11]

This paper emphasizes another topic named "information distribution." Design and development are a series of decisions. Each decision results in information about the system which can be used in making later decisions. We want eventually to discuss the distribution of that information among those working on the system and to deal with its organization in documentation. To prepare for this discussion we deal first with (1) the concept of system structure, (2) constraints on the order of decisions, and (3) some observed characteristics of good programmers.

STRUCTURE DEFINED

The word "structure" is used to refer to a partial description of a system. A structure description shows the system divided into a set of modules, gives some characteristics of each module, and specifies some connections between the modules. Any given system admits many such descriptions. Since structure descriptions are not unique, our usage of "module" does not allow a precise definition parallel to that of "subroutine" in software or "card" in hardware. The definitions of those words delineate a class of objects, but not the definition of "module." Nevertheless, "module" is useful in the same manner that "unit" is in military or economic discussions. We shall continue to use "module" without a precise definition. It refers to portions of a system indicated in a description of that system. Its precise definition is not only system dependent but also dependent upon the particular description under discussion.

The term "connection" is usually accepted more readily. Many assume that the "connections" are control transfer points, passed parameters, and shared data for software, wires or other physical connections for hardware. Such a definition of "connection" is a highly dangerous oversimplification which results in misleading structure descriptions. The connections between modules are the assumptions which the modules make about each other. In most systems we find that these connections are much more extensive than the calling sequences and control block formats usually shown in system structure descriptions.

The meaning of the above remark can be exhibited by considering two situations in which the structure of a system is terribly important: (1) making of changes in a system, and (2) proving system correctness. (I feel no need to argue the necessity of proving programs correct, or to support the necessity of making changes. I wish to use those hypothetical situations to exhibit the meaning of "connection.")

Correctness proofs can become so complex that their own correctness is in question [e.g., 14, 15]. We would like to simplify the proofs by using the structure of the program, proving the correctness of each module separately. For each module we will have a set of hypotheses to prove and a description of the module. In our hypotheses we can distinguish the things we expect a module to accomplish from the things which we assume other modules will guarantee. Those statements are the connections between the module being examined and the rest of the system. The proof process will be facilitated only if the amount of information in the hypotheses is significantly less than the amount of information in the full description of the connected modules. In the extreme case, where one module's correctness is predicated upon the complete description of another module, the proof of the first module's correctness will be as complex as if the two were considered a single module.

We now consider making a change in the completed system. We ask, "What changes can be made to one module without involving change to other modules?" We may make only those changes which do not violate the assumptions made by other modules about the module being changed. In other words, a single module may be changed only as long as the "connections"

still "fit" Here, too, we have a strong argument for making the connections contain as little information as possible.

FACTORS INFLUENCING THE ORDER OF DECISION MAKING

Progress in a design is marked by decisions which eliminate some possibilities for system structure. The fact that those possibilities have been eliminated can be part of the rationale for subsequent decisions. If the information is used, the order of decision making (in time) affects the structure of the resulting product. Examples of interest can be found in [4]. We can identify three considerations, each suggesting a partial ordering on the decisions.

1. Obtaining 'good' external characteristics.

All systems have characteristics which are not pleasing to the users. Usually they were not determined by explicit deliberations; they were the unnoticed implications of decisions about other aspects of system structure. To consistently avoid such errors we can make the decisions about external characteristics first. We use the resulting information to make the later decisions. The internal decisions would be either derived from or checked against the complete specifications of the external factors. This is the basis of the "top down" or "outside in" approach discussed in [1, 2, 3, 4].

2. Reducing the time interval between initiation and completion of the project.

Competitive pressures may require the use of large groups to produce a system in a sharply limited period of time. Additional men speed up a project significantly only <u>after</u> the project has been divided into subprojects in such a way that separate groups can work with little interaction

(i.e., spending significantly less time in inter-group decisions than in intra-group decisions). This consideration affects the order of decisions in that it encourages very early splitting of the system into modules which are then designed completely independently. The desire to make the split early and "get on with it" encourages a splitting along familiar lines and in agreement with existing personnel classifications.

Time pressures encourage groups to make the split before the externals are defined. Consequently we find some adverse effect on the useability of the product. Haste also makes poor internal structure more likely.

3. Obtaining an easily changed system.

Systems are changed after construction either because their original characteristics proved insufficient or because another application was found. We have already noted that the difficulties in changing systems are related to the assumptions which each of the modules makes about its environment. Since each decision is usually made on the assumption that the previous decisions will hold, the most difficult decisions to change are usually the earliest. The last piece of code inserted may be changed easily, but a piece of code inserted several months earlier may have "wormed" itself into the program and become difficult to extract. These considerations suggest that the early decisions should be those which are the least likely to change; i.e., those based on "universal" truths or reasoning which takes into account little about a particular environment. The remaining facts must be used eventually, but the possibility of change suggests using the most general information first.

Since such external characteristics as job control language and file commands are very frequently changed, the "outside-in" approach may make the system harder to change. Further, those decisions which should be made early on this basis are not usually those which allow the project to be quickly subdivided into independent assignments. As a rule, decisions which do not use all the available information about a system (i.e., the general decisions) take more time.

In summary, each of the three considerations suggests a partial ordering of the decisions. Those orderings are usually inconsistent in that it will be impossible to satisfy them simultaneously.

DOCUMENTATION SYSTEMS

For any complex system there must be documentation about the system for use by the human beings who must complete it. Programs and wiring diagrams do completely define the algorithm which they will execute, but this form of documentation is not usually appropriate for people. Consequently there are always papers which attempt to answer the questions most likely to be asked. There is usually no attempt to make the documentation complete (i.e., equivalent to the code) for software, thus certain questions must be answered by reference to the code.

When a system is strongly connected, this documentation must be read by persons not closely involved with the module being documented. Because each working group develops a unique module organization and a corresponding set of concepts and terms, the documents which they write are difficult for outsiders to read.

The natural response is to require all documentation to be written with a standard organization and vocabulary [16]. A standard is made company-wide to allow anyone in the organization to find some piece of information without needing to learn the concepts and vocabulary peculiar to one system or module.

Such approaches raise several questions:

1. Is it really desirable to have all information equally accessible to all in the company (or project)?
2. What is the effect of documentation standards on the resulting system?
3. What is the result of a non-standard system being described using a standard document organization?

Documentation standards tend to force system structure into a standard mold. A standard for document organization and vocabulary makes some assumptions about the structure of the system to be described. If those assumptions are violated, the document organization fits poorly and the vocabulary must be stretched or misused. Consider the following example. In most operating systems there exists a module which handles all job control statements from the time they are read in until the job is completed. As a result, most documentation systems can insist that there be a section describing such a module. Now consider an organization such as that of the T.H.E. system in which there is no such module because most of the processing is handled in modules which are also used for other purposes. If we adhere to the documentation standard we will duplicate information and describe one module in the documentation of another.

If there are to be standard documentation organizations, they must be designed to make the minimum number of assumptions about the system being documented. If so, they will be of little help in making the document readable to people who do not understand the structure of the system.

ON SOME PROPERTIES OF GOOD PROGRAMMERS

The following observation is essential to the remainder of this paper:

"A good programmer makes use of the usable information given him!"

The good programmer will try to use his machine well. He is actually programming for a "virtual machine" defined by the hardware and his knowledge of the other software on the machine. His training and his nature lead him to make full use of that extended machine.

Sometimes the uses are obvious. The programmer makes use of a subroutine from some other module, or a table of constants already present for some other piece of code. Sometimes these uses are so marginal as to be laughable, e.g., the use of a 3-instruction subroutine or the borrowing of a single constant. In the terms of our previous discussions, such extreme cases increase the connectivity of the structure without appreciably improving its performance.

Sometimes the uses are less obvious. For example, a programmer may make use of his knowledge that a list is searched in a certain order to eliminate a check or an extra queue. In the area of application programming we may find a programmer who introduces an erroneous value for Π knowing

that because of an error in the sine routine the erroneous value will cause his program to converge more rapidly.

Such uses of information have been so costly that we observe a strange reaction. The industry has started to encourage bad programming. Derogatory names such as "kludger," "hacker" and "bit twiddler" are used for the sort of fellow who writes terribly clever programs which cause trouble later on. They are subtly but effectively discouraged by being assigned to work on small independent projects such as application routines (the Siberia of the software world) or hardware diagnostic routines (the coal mines). In both situations the programmer has little opportunity to make use of information about other modules.

Those that remain (the non-bit-twiddlers) are usually poor programmers. While a few refrain from using information because they know it will cause trouble, most refrain because they are not clever enough to notice that the information can be used. Such people also miss opportunities to use facts which should be used. Poor programs result. Since even a poor programmer sometimes has a "flash of brilliance" (e.g., noticing that two bytes in a control block can be simultaneously set with one instruction because they are adjacent and in the same word) we still have no control of the structure.

We have found that a programmer can disastrously increase the connectivity of the system structure by using information he possesses about other modules. We wish to have the structure of the system determined by the designers explicitly before programming begins, rather than inadvertently

by a programmer's use of information. Consequently, we discourage the bit twiddlers and pay a price in poor programming without obtaining complete control of the structure.

THE USE OF DESIGNER CONTROLLED INFORMATION DISTRIBUTION

We can avoid many of the problems discussed here by rejecting the notion that design information should be accessible to everyone. Instead we should allow the designers, those who specify the structure, to control the distribution of design information as it is developed.

Our concerns about the inconsistent decision orderings were based on the assumption that information would be used shortly after the corresponding decision. The restrictions placed by the three considerations are considerably relaxed if we have the possibility of hiding some decisions from each group. For example, we have noted a conflict between the desire to produce an external specification early and the desire to produce a system for which the external interface is easily changed. We can avoid that conflict by designing the external interface, using it as a check on the remaining work, but hiding the details that we think likely to change from those who should not use them.

If we want the structure to be determined by the designers, they must be able to control it by controlling the distribution of the information. We should not expect a programmer to decide not to use a piece of information, rather he should not possess information that he should not use. The decision is part of the design, not the programming.

Reflection will show that such a policy expects a great deal from the designers. We currently release all the information about a module; to do so is considerably easier than (1) deciding which information should be released and (2) finding a way of expressing precisely the information needed by other modules. Preliminary experience has shown that making appropriate definitions is quite difficult. Acquiring skill in making those definitions is vital because we will be able to successfully build systems while restricting programmers' information only if we learn to provide them with precisely the information they need.

EXAMPLES

I believe it worthwhile to give some concrete examples of information which is now widely disseminated within a project and should instead be sharply restricted.

1. Control Block Formats

Every system contains small amounts of information in pre-formatted areas of storage called control blocks. These are used for passing information between modules and are considered to be the interfaces. For this reason formats are usually specified early in the project and distributed to all who are working on the project. The formats are changed many times during the project. Few programmers on any project need to know such formats. They need a means for referring to a specific item, but not more. They need not even know which information is grouped into one control block.

2. Memory Maps

It is common to begin a description of an operating system by (1) describing the main modules and (2) showing how the core storage is divided among those main modules. Soon there is a complete map of the memory showing how that resource is allocated. Reasonably sophisticated designers show the borders of allocated areas as symbolic rather than absolute addresses, but the order of memory assignment is specified. Only a small portion of this information derives from hardware decisions. There is no legitimate way to use the map information. It would be frightening if someone developed code that would not work if the map were changed. Such maps are almost invariably changed because something which was fixed becomes variable or vice versa. The information is only needed at assembly time. We could survive if it were input to the assembler and not known by anyone else.

When there is a virtual memory or other mechanism for swapping built into the system, the distinction between resident and non-resident items should not be broadcast. If there are several kinds of core storage, the allocation of modules and data among those storage types should not be known to those who are writing the modules. If partial preloading of certain programs is envisaged, the decision as to which modules will be preloaded should be hidden. Each of these decisions is worthy of attention, but few should know the result.

3. Calling Sequences

Calling sequences are the secret hobby of every system programmer. We begin to look at new hardware by inventing a calling sequence. Throughout

the design and implementation, the calling sequence is simplified, generalized, made more efficient, etc. Each time we face a decision. Either modules all over the system are altered or the new sequence is added to a growing set of calling sequences. In the latter case generating a call to a routine requires determining which sequence it uses.

Most routines can be written, and written well, without knowledge of the calling sequence if the programmer is provided with a programming tool which allows him to postpone decisions about register allocation for parameters, return addresses, and results. Such features can be provided in an assembler with macro facilities.

4. JCL Formats

One characteristic which should be easy to change is the syntax of the so called Job Control Language, the means by which the user describes his job's gross characteristics to the operating system. The design of a JCL implies assumptions about the way that the system will be used which may later prove to be false or too restrictive. There exist systems in which JCL format information has been used so much that reasonable changes are beyond the scope of the usual organization. Often changes require user provision of duplicate information and/or the maintenance of duplicate tables. (See, for example, [17].)

Most of the people working on an operating system need very little knowledge about the JCL. The only people who need to know the format are those who are writing the syntax analyzer for the language.

5. Location of I/O Device Addresses

It is widely recognized that device addresses should not be built into code but stored in tables associated with each job. However, it is usual that all programmers are given knowledge sufficient to allow them to find and use the table. For example, many modules will send messages to a user at his teletype. If later one wishes to intersect those messages and reinterpret or suppress them for a special class of users, the job is horrendous. Most programs did not need that information. Access to a module which would send messages for them is sufficient.

6. Character Codes

Some hardware information should not be released. I have seen one compiler in which the association made by the hardware between card characters and integers was so widely used that a second version of the compiler (for a new machine) contained a module which translated from the new character code to the old one and back again.

The efficiency gained by using the character code information (e.g., by using arithmetic tests to determine if a given character is a delimiter) is often not worth the price paid. Where it is worthwhile, the knowledge can be closely restricted <u>if</u> the designers pay attention to the problem. Certainly the decision to use or not to use the information should not be left up to an individual programmer.

CONCLUSION

The inescapable conclusion is that manufacturers who wish to produce software in which the structure is under the control of the designers, must develop a documentation system which enables designer control of the distribution of information. Further, they must find and/or train designers who are able to define or specify modules in a way which provides exactly the information that they want the programmers to use. Until we can completely staff a project with men who have the intellectual capacity and training to make that decision for themselves, some must make the decision for others. An assembler which allows the insertion of some hidden information at "assembly time" will aid in maintaining efficiency.

I consider the internal restriction of information within development groups to be of far more importance than its restriction from users or competitors. Much of the information in a system document would only harm a competitor if he had it. (He might use it!)

It is worth repeating that the decision about which information to restrict is a design decision, not a management one. The management responsibility ends with providing the appropriate information distribution mechanism. The use of that mechanism remains a design function because it determines the structure of the product.

APPENDIX

A MODULE DOCUMENTED ON THE BASIS OF "KNEED TO KNOW"

INTRODUCTION

Assume the system under construction to be a translator for string manipulation algorithms based upon Markov Algorithms. Such a package must contain a representation of the variable length string known as the register which constitutes the only memory in a hypothetical Markov Algorithm machine. Assume further that the decision has been made that the knowledge of this representation be confined to a single module in spite of the fact that almost all actions done by the system will involve changes in the register. The purpose of this decision is to make the representation easy to change.

The statements which follow provide all the documentation of such a module which should be available to its users. They are intended to provide all the information necessary to use the module, i.e., to manipulate the register, yet no information about the representation of the register in the machine. The method used is to define five procedures, to specify their initial values if they are functions, to specify the type of their parameters where they have parameters. Further, a statement is made as to the effect of a call on the procedures on the values of the other functions in the package. This is done by indicating the new value of any changed functions as a function of their old values and the values of parameters to the called procedure. A value before the change is shown enclosed in single quotes (e.g., 'length'). Values after the change are

shown unquoted. The actions which take place in the event of errors are specified to be procedure calls. It is assumed that should such a call occur, (1) no values will have been changed, and (2) upon a return from the procedure called, the attempt to perform the routine specified will be repeated completely.

DEFINITIONS

INTEGER PROCEDURE: LENGTH
possible values: an integer $0 \leq$ length ≤ 1000
effect: no effect on values of other functions
parameters: none
initial value: 0

INTEGER PROCEDURE: GETCHA (I)
possible values: an integer $0 \leq$ GETCHA ≤ 255
parameters: I must be an integer
effect: no changes to other functions in modules

if $I \leq 0 \vee I >$ LENGTH then a procedure call to a user written routine RGERR is performed. (program cannot be assembled without such a routine)

initial value: undefined

PROCEDURE: INSAFT(I, J)
possible values: none
parameters: I must be an integer
J must be an integer
effect:

if $I < 0 \vee I >$ 'LENGTH' $\vee J < 0 \vee J > 255$ then a subroutine call to a user written routine INSAER is performed. (routine required)

else LENGTH = 'LENGTH' +1 if LENGTH ≥ 1000 a subroutine call to user written function LENGER is performed.
GETCHA(K) =
if k $\leq$ I, 'GETCHA(I)'
if k = I+1, J
if k > I+1, 'GETCHA(K-1)'

PROCEDURE: DELETE (I, J)
possible values: none
parameters: I, J must be integers
effect:

if $I \leq 0 \vee J < 1 \vee I+J >$ 'LENGTH' +1 then a procedure call to a user written routine DELERR is performed.

else
LENGTH = 'LENGTH' - J.

GETCHA(K) = if k < I then 'GETCHA(K)'
if k $\geq$ I then 'GETCHA(K+J)'

PROCEDURE: ALTER(I, J)
possible values: none
parameters: I, J must be integers
effect:

if $I \leq 0 \vee I >$ 'LENGTH' $\vee J < 0 \vee J > 255$ then a subroutine call to a user written routine ALTERERR is performed.

GETCHA(K) = if K $\neq$ I then 'GETCHA(K)'
if K = I then J

DISCUSSION

It is possible to verify the completeness of these definitions by showing that a value is defined for each function for every possible sequence of calls. The possibility of infinite looping through repeated calls of error routines exists, but this would be an error in usage not in definition.

One can demonstrate that a minimum of information is given out by the definitions by showing first its sufficiency for use (i.e., completeness) and by showing that the widest conceivable variety of implementations can fit the definitions.

The usual form of documentation would be (1) much more wordy, (2) more revealing of internal aspects. In fact, because natural language is used the completeness can only be assured by exhibiting the internal structure.

The mnemonic names used here carry no essential information. They could be replaced by 'x1', 'x2', etc. at no theoretical cost, but at the practical cost of being obscure.

The definitions are obscure now to a reader unfamiliar with the register of a Markov machine. This can be alleviated by a supplement suggesting ways to use the functions (e.g., a teaching supplement) having no official status.

References

1. Parnas, D. L. and Darringer, J. A., "SODAS and A Methodology for System Design," Proc.AFIPS 1967 Fall Joint Computer Conference, pp. 449-474.

2. Zurcher, F. W., Randell, B., "Multilevel Modeling - A Methodology for Computer System Design," Proc. 1968 IFIP Conference.

3. Parnas, David L., "More on Simulation Languages and Design Methodology for Computer Systems," Proc. SJCC 1969, pp. 739-743.

4. Dijkstra, E. W., "Notes on Structured Programming," publication of the Technical University of Eindhoven, The Netherlands.

5. Dijkstra, E. W., "Complexity Controlled by Hierarchical Ordering of Function and Variability," in Software Engineering, proceedings of a meeting at Garmisch, Germany, October 7-11, 1968.

6. Gill, Stanley, "Thoughts on the Sequence of Writing Software," in Software Engineering, proceedings of a meeting at Garmisch, Germany, October 7-11, 1968.

7. Dijkstra, E. W., "Structured Programming," in Software Engineering Techniques, proceedings of a meeting held in Rome, October 27-31, 1969.

8. Dijkstra, E. W., "A Constructive Approach to the Problem of Program Correctness," BIT 8, vol. 3, 1968.

9. Naur, P., "Proof of Algorithms by General Snapshots," BIT 6, 1966.

10. Wulf, et al, "BLISS Users Manual," publication of the Carnegie-Mellon University, Pittsburgh, Pa., USA.

11. Waite, V. M., "The Mobile Programming System: STAGE 2," CACM 13,7 (July, 1970), pp. 411-421.

12. Parnas, David L., "On the Use of Transition Diagrams in the Design of A User Interface for an Interactive Computer System," Proc. 1969 National ACM Conference, pp. 379-386.

13. Hartman, P. H., Owens, D.H., "How to Write Software Specifications," Proc. 1967 FJCC, pp. 779-790.

14. Balzer, Robert M., "Studies Concerning Minimal Time Solutions to the Firing Squad Synchronization Problem," PH.D. thesis, Carnegie Institute of Technology, 1966.

15. London, R., "Certification of Treesort 3," CACM (June, 1970).

16. Selig, F., "Documentation Standards," in Software Engineering, proceedings of a meeting at Garmisch, Germany, October 7-11, 1968.

17. Braden, et al, "An Implementation of MVT," publication of the University of California at Los Angeles.