TEACHING THE COOPERATIVE PRODUCTION OF PROGRAMS

D. L. Parnas

Rapporteurs: Dr. P. Henderson
Mr. R. Kerr
Mr. D. Wyeth

1.0 Introduction

This is a report on a course entitled "Software Engineering Methods", which has been taught to undergraduate students at the Carnegie-Mellon University during the last two academic years. The course is "project oriented" and aims to educate by providing experience in the use of techniques. This report describes both the structure of the course and some of the material taught.

1.1 Cooperative Programming Defined

The meaning of cooperative programming is similar to that of software engineering, better illustrated as a contrast to solo programming. Whereas in solo programming a single person constructs a program which will not be touched by other people, the essential characteristic of cooperative programming is that many people are involved with the product which we refer to as software. Several people may cooperate in producing it and/or it is used or modified by persons other than the original author.

Another characteristic of software is that one is considering not a single program but a family of programs. The word "family" here is used in the same sense that System 360 is a family of computers. Although two computers of the 360 range may be physically very different, they can be described by the same programmers manual.

1.2 Aim of the Course

The first part of this paper concentrates on:

1. Defining what we want to teach the students to do in terms of the results we want them to obtain.
2. Explaining the primary skills which the students must acquire.
3. Discussing the teaching methods and course organisation.

A deeper exploration of some of the techniques taught is contained in the second and third sections of this paper.

The type of results we hope to obtain are best illustrated by describing one of the projects undertaken as part of the course. The aim of this project is to construct a family of programs to produce a KWIC index. The project is divided into five work assignments or modules. Each of 20 students did one work assignment resulting in four versions of each module. The idea was that any combinations of one version of each of the five modules would form a working KWIC index system. If completely successful, there would have been 1024 working KWIC index programs in the family of programs.

TABLE I

Version	Assignment 1	2	3	4	5
A	OK	OK	OK	NOT COMPLETED	OK
B	OK	OK	OK	OK	INCORRECT
C	INCORRECT	STUDENT DROPPED	NOT ASSIGNED	OK	INCORRECT
D	INCORRECT	OK	OK	NOT ASSIGNED	OK
E	NOT ASSIGNED	OK	OK	OK	NOT ASSIGNED

Table I gives the versions of each module which we judge correct. Each version was not proven correct but was tested individually to establish some degree of reliability. From the Table, we may calculate that there were 192 working combinations. We could not test all of these. An experiment was planned to test 25 combinations so that (1) each version was used in at least two combinations and (2) each version was in at least one combination where it was the only difference from another tested combination. The versions were combined and tested by someone who knew nothing of the project thus demonstrating the case of integration.

TABLE II

Combination tested					Execution Time (sec) (excludes compilation of 6-8 sec)
1A	2B	3B	4B	5A	37.26
1A	2D	3D	4B	5A	11.42
1A	2D	3A	4C	5A	10.87
1B	2E	3A	4C	5A	10.31
1A	2E	3A	4B	5D	8.53
1B	2A	3E	4C	5B	21.79
1A	2A	3B	4B	5B	302.99
1A	2A	3B	4B	5A	50.16
1A	2A	3B	4C	5A	36.69
1A	2A	3D	4C	5A	11.07
1A	2B	3D	4C	5A	10.99
1A	2A	3B	4E	5A	43.30
1A	2D	3B	4E	5A	43.61
1A	2D	3B	4E	5D	19.17
1A	2E	3B	4E	5D	19.16
1A	2E	3B	4C	5D	28.48
1A	2B	3B	4C	5D	27.23
1A	2B	3D	4C	5D	8.43
1A	2B	3D	4C	5B	76.34
1A	2B	3D	4B	5B	113.32
1A	2B	3B	4C	5B	238.88
1A	2B	3E	4C	5D	10.06

Table II describes the results of this experiment and illustrates the type of variety obtained. The variation in execution times is accounted for partly by the constraints placed on different versions of a given assignment e.g. conservation of space, conservation of time, etc. It should be pointed out, however, that the fastest program also used the least space! All the modules marked as believed correct in Table I did work correctly when combined and tested.

The following features are significant in terms of the results we are trying to achieve.

1. We succeeded in getting programs with interchangeable yet non-identical parts.
2. Once the modules were specified, there was no communication between the module authors. The students did not know which combinations of modules would be tested and thus which versions their module would have to cooperate with. For this reason, they could use no information other than the published specification and thus could take no short cuts.
3. In this and in other similar experiments, an unexpected advantage was that when a particular combination didn't work, one could determine unambigously which module was in error.

4. Integration and testing was carried out by a graduate student who knew nothing about any module implementation.

5. A further advantage of this method is that one can study one part of a system with minimal knowledge about the rest. Major changes to the system can be confined to one part. Major decisions such as whether to do all the alphabetization at once or alphabetize a little at a time in parallel with printing can be altered by changing only one module. Similarly, the decision whether to store all the data in core or part of it on disk also involves only one module. All the major decisions of this example were handled in such a way that at most one part would have to be changed in altering a decision.

1.3 Course Organization

The philosophy behind the course is that it is better to teach methods of problem solving rather than to teach known solutions to specific problems. The course may be thought of as divided into three phases based upon the type of assignment given to the students. In the first phase, the assignments consist of introductory small projects. The students are given definitions of relatively small devices common in software engineering (e.g. a stack, queue or tree structure). For each object some are asked to produce implementations while others are asked to write small programs which use the object. Programs are exchanged and tested. Students are also taught to write specifications.

In the second phase of the course, the class builds a "family" of small systems from a design presented to them by the instructor. The project is a small scale system but larger than the previous projects. Such a project is the KWIC index system described above. The system is divided into approximately six modules. Each module is given a precise definition and each student builds a version one module.

In the third phase of the course another system is started. In this one the students are given only a rough picture of what the system is intended to do. The class, working as a design committee or a system committee, goes through the exercise of squeezing the real intentions of management (in this case the instructor) from the vague descriptions and conversations, producing a more precise structure such as was given to them in the second phase. They then go through the exercise of dividing the system into modules, providing precise definitions of the modules, and (if time permits) completing the system as in the previous project.

Throughout all three phases the lectures are coordinated with the projects so as to explain to the students what they are doing and why they are doing it. The initial lectures are used to give the students sufficient information to enable them to complete the small projects and start the KWIC index project. The ground covered includes the construction of a module from specifications which satisfies the specifications and is free of misinterpretations and writing small module specifications themselves.

During the second phase the project's design is motivated for the students. They are shown how the system's decomposition into modules was arrived at, they are given the reasons for defining the interfaces chosen, and are shown some alternative formulations together with the relative advantages and disadvantages. We examine possible implementations of each module, taking care to show several alternatives and show situations in which each is preferred. The KWIC index system is intentionally not the best known design so that students can suggest improvements to the design. In preparation for the final project the topics covered include definition of interfaces, the writing of specifications as opposed to the reading of them, and the verification of specifications (which is just as difficult as proving programs correct). The completion of the KWIC index project is timed for the end of these lectures enabling the project to be discussed in class.

The aim of the final project is to expose students to the problems of design and to increase their ability to look at a design and see how much of the complexity in a system is intrinsic complexity and how much is due to early design errors. Experiencing their own design errors and those of other students is found to have a much greater impact than merely showing examples of successful projects. During this phase of the course the students learn how to design systems and how to improve the design of others.

Currently this is a one semester course (48 hour course - 3 lectures a week for 16 weeks) though the second project is rarely completed. It is hoped to correct this by adding two hours of laboratory work a week when the students will be given programming help. Past courses have shown that it is usually the students' lack of programming experience and proper training which prevents their completing the course successfully.

1.4 Structure

The course is successful because it is founded upon a precise concept of "structure". The structure of the family of KWIC index programs can be written down as a specification of the five modules.

The remark is often heard that the "structure of a system is good but the implementation is terrible" implying that the "structure" is separate from the "implementation". This discussion of structure is aimed at showing that structure and implementation have a close connection, that a careful and considered design of structure is necessary, and that implementation must be carefully controlled if it is not to determine the structure.

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. A module must be loosely described as "part of a system". It refers to the 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 consideration. In the above discussion of the course, a work assignment corresponds to a module. A module may itself be further subdivided. We specifically caution against attaching further meaning to the word "module". Specifically assembly module and memory load module are not intended.

In a structure description of a system the modules are normally correctly identified, but the connections between the modules are often oversimplified and therefore inaccurate. Many assume that the "connections" are control transfer points, passed parameters and shared data etc. 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. The oversimplification results from ignoring other assumptions made by one module about another. This can be illustrated in two ways. Consider two situations, one in which the system is to be proved correct and another in which a change is to be made to the system, and ask the question "of what help will it be to us to divide the system into modules?"

Correctness proofs for large programs can become so complex that their own correctness is in question. For large systems we must make use of the structure of the programs in producing the proofs. We must examine the programs comprising each module separately. To prove each module correct separately, we must make some assumptions about the other modules. That set of assumptions is the set of connections between the modules. The task of proving system correctness will be facilitated by this process only if the amount of information in the assumptions is significantly less than the information in the complete description of the programs which implement the module.

We now consider making a change to the complete 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 while the "connections" still "fit". Here too we have a strong argument for making the connections contain as little information as possible.

The information of concern here is the information being used at program design time rather than at run time. We want to conserve the amount of information passed between programmers at the time of writing, but not the amount of information passed between modules at run time.

This point can be demonstrated by an example. Consider an operating system which includes a drum/disk handling module and a module whose job it is to wake-up processes. Attached to the drum/disk handling module is a queue. Each queue element contains two items, one is the name of the process that wants a page and the other is the name of the page required. The drum/disk handling module reads this queue and separates the two items of each element. It keeps the names of the required pages and passes to the other module the names of the processes in the queue. Thus only half the information contained in the queue is passed on at run time. The assumption made here is that the drum/disk handler is going to handle requests in the order first come first served because it passes on names of processes in the order in which they come in. This assumption could turn out to be unfortunate. Every time a page is read in the wake-up module wakes up the next process on its queue of processes. The

unfortunate thing about this design is that one is no longer free to change the drum/disk handler so that it tries to reduce arm movements. This example shows that by reducing the amount of information passed between modules at run time the connectivity (i.e. assumptions between modules) has been increased by sharing the assumption of the FIFO servicing of page requests between two modules. Thus to change that assumption, one cannot restrict the change to one module but must spread it **across** both modules.

This explains why it is important that we can write down the structure of the KWIC index system. The set of specifications contain all the assumptions that one student is allowed to make about another students work. These specifications are the only thing all members of the family must have in common. It is because a formal, well defined, and accurate description of the structure can be written down that this approach to the course is successful.

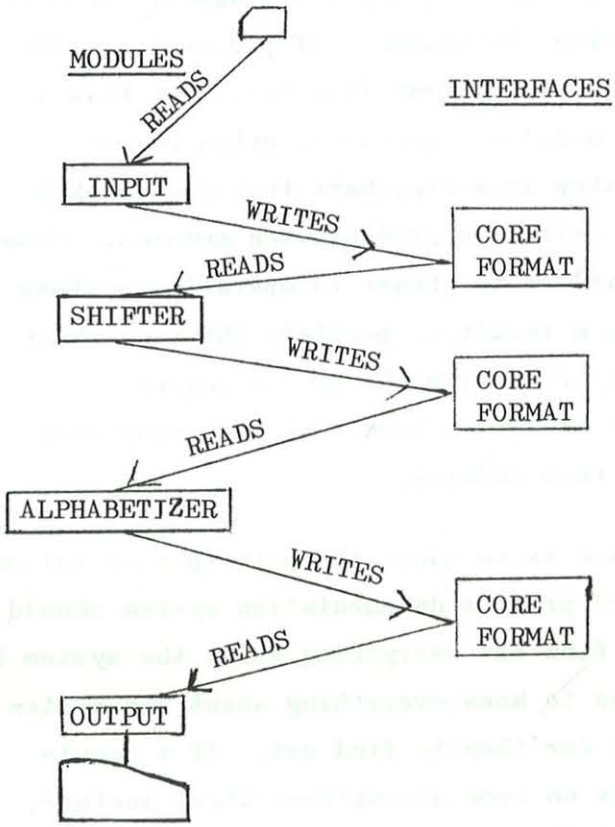
2.0 Identification and Specification of work assignments

The first section presented the notion of the structure of a program as a partial description of that program showing it divided into a set of parts and showing the assumptions which connect these. These assumptions are predicates which one could use to prove parts of the program correct or to decide whether alterations can be localised to single modules.

The term modular programming requires clarification in the context of these lectures. Many people regard modular programs simply as programs which are split up into lots of little modules. Unfortunately, many systems which exemplify things we ought not to do are modular in this sense. The concept of modularity goes beyond the simple act of subdivision and the course attempts to teach principles for decomposition and specification.

Decomposition and specification are issues in solo programming. They are aggravated in the multi-person situation. Henderson and Snowdon [2] give a nice example of what can happen when we are not very precise with ourselves. The problem of communicating with ourselves is minor compared with that of communicating with others.

In the process of decomposition, we have to decide how to divide our systems into modules and not just arbitrarily cut them up according to what seems convenient from an organizational point of view. This point is illustrated by comparing two methods of designing the KWIC program quoted in the first lecture. When many people were asked how they would decompose the KWIC program there emerged a remarkable consistency in their replies.



A card format would be designed which would serve as input to an INPUT module. The INPUT module would write its data as a core table, necessitating the design of a core format. The information held in core would be input to a CIRCULAR SHIFTER module which would produce a new table in core (or alternatively a directory referencing the old table). An ALPHABETIZER would read the table generated by a CIRCULAR SHIFTER and prepare yet another core table which would ultimately be processed by an OUTPUT module.

The deficiencies in this type of design become obvious when we consider the effects of making alterations. Amendments necessitated by a change in the input format would be confined to the INPUT module

but alterations to its output CORE FORMAT would have consequences for every other module. It is not only changes to data structures which can have widespread effects but also alterations to algorithms. If we distribute the alphabetization process over other operations as in Hoare's FIND [3] we must amend both the ALPHABETIZER and OUTPUT modules and possibly others.

That there was great consistency in approach to the design of the KWIC program can be explained by studying the diagram. This is, in fact, a flowchart displaying the sequence of processing. The first thing programmers are taught is to draw flowcharts but this is not the correct way to define modules. One is so often passing information from step to step in a flowchart that one quickly establishes formats which will be shared between modules. Some of the things which are hardest to change in operating systems such as OS and TSS 360 are a result of people's thinking about the design of the system by following the processing of a particular element through it. They then took the major steps in the processing and called them modules.

A better design approach is to adopt the principle of information hiding. The goal of a good program documentation system should not be to enable everybody to find out everything about the system but to ensure that nobody needs to know everything about the system and, even, to make it difficult for them to find out. If a module depends for its correctness on some assumptions about another, this increases the connectivity of the structure when what we want is loosely connected structures. Many good programmers attempt to use every single piece of information about both hardware and software in order to produce efficient, tightly coded programs. They often use information which they should rely on to remain true, e.g. Fred does not use bit 17 in the control block. It is very much a matter of experience deciding which information one can use and which one should not. Documentation which does not "broadcast" all information removes the decision from the inexperienced programmer and leaves the decision to the designers who controls the information distribution.

A description of an improved subdivision of the KWIC program follows. This contains modules of the same name as in the first approach but providing different functions. A new module LINE HOLDER is introduced. This provides functions for storing and retrieving in the core tables items which are now addressed by line number, word number and character number. The core format and storage and retrieval techniques are now hidden in the LINE HOLDER. If the core format has to be altered, only the LINE HOLDER module requires amendment. The CIRCULAR SHIFTER appears similar to the LINE HOLDER but acts as a read-only memory which looks like the LINE HOLDER but contains the circular shifts not through lines. One can access information from this without knowing how it is represented. The ALPHABETIZER now provides a set-up function ALPH and the function ITH(i) which gives the index of the i^{th} alphabetical line. This completely hides the sorting technique used and even the time at which sorting takes place. Versions of ALPH and ITH have been implemented in which ALPH did practically nothing and alphabetization took place during calls on ITH. Other versions in which ALPH did most of the work and ITH was simply represented by an array have also been produced. The important point is that, by looking at the specification, one cannot tell which of these one has.

The course attempts to teach the students to make real design decisions and to make the most solid ones, those which will hold for the whole family of modules, first. Other decisions are postponed to the modules they concern. This process tends to hide information. Attempts are made to teach the students the type of decision which should be hidden and how to hide them. Discussions of examples showed this policy to be in conflict with current programming practice. It is popular practice, for example, to design calling sequences for subroutines early on. Should the decisions regarding register allocation etc. be altered, changes must be made everywhere the subroutine is referred to. 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 and return addresses. Such features can be provided by macro facilities.

Having decided what the modules will be, it is now necessary to specify them. Since the modules will be written by different people, it is necessary to state the functions the modules will perform and Natural language descriptions are inadequate for this purpose. A more formal specification is required. A black box specification is desirable and it should not give information which is later denied. If we describe a sorting program by an algorithm, the algorithm will specify one particular course of action (exchange or leave as they were) when it encounters an identical pair of items although we may not really care what happens in this case. It is difficult to describe what we do not care about. In the "equivalent algorithm" approach, the algorithm may be equally complicated or more complicated than the program. It gives much more information than is needed and that raises problems of misinterpretation. As much irrelevant information as possible should be removed in order that the programmer can concentrate on the important material.

We wish to establish a set of identities between the functions which each module provides. In specifying a square root procedure, we could describe an iterative numerical algorithm which ultimately converges to the square root. This would be a horrible way to describe what the purpose of the procedure really is. We could instead simply state the relationship:

$$|(SQRT(X))^2-X|<\epsilon$$

It is important to be able to give this sort of definition for software modules.

A notation is presented which, although lacking some element of formal definition, is sufficient for the practical work described so far. It has a somewhat ALGOL-like appearance and, for each module, describes the effects its functions have on one another. For example, a stack mechanism can be described by four functions F1(a), F2, F3 and F4. After a call on F1, we can state that

F3 = a

F4 = (old value of F4)+1.

We also state

(:) F1 followed immediately by F2 has no net effect on the state of the module

(ii) F3 has no effect

(iii) F4 has no effect on the module state

From these statements one may conclude F1; F3; F4; F2, F1; F1; F2; F3; F2 etc. have no net effect.

F1 is of course PUSH, F2 is POP etc. This type of specification is quite adequate to describe the function of the stack without concerning itself with pointers, links and such implementational irrelevancies. Since the details of implementation are not presented, there is no danger that writers of other modules will interfere with the pointers.

[4] shows the specification of a binary tree. Briefly, this provides functions:

FA = father, LS = left son, RS = right son, SLS = set left son, SRS = set right son, VAL = value, SVA = set value. There are, in additon, several predicates for testing for the existence of various tree members. Taking SLS(i) as an example, its specification specifies the following relationships and actions. FA(i) must exist and LS(i) must not. A value of k is defined such that the old value of FA(k) was undefined and after SLS(i) is completed LS(i) = k and FA(k) = i.

The value of this type of specification is that it fully describes the function of the tree module but never specifies what values will be given to LS and RS. The tree could be implemented either as a linked list or using an array as in Floyd's Treesort [5] but this specification would be valid in both cases. The specification is sufficient for someone else to use the tree module without being aware of the nature of the implementation.

The course tries to teach people how to make these abstractions. Abstract is not a euphemism for vague. These specifications are very precise; they can be used to prove theorems about the modules.

It is not possible to express this type of specification nearly in a programming language. This is because it is not individual program steps which are being specified. Much can be omitted which would have to be included if program steps were being described. If a module causes some value to change it is not necessary to say how it is changed. The notation becomes familiar to students and after that they do not object very much.

3.0 Response to detected errors

That computer programs, even well structured programs, will not always perform correctly is a fact of life. Thus is is important to teach how to cope, at the design stage, with the possibility of run time errors. It is useful therefore to be able to classify errors and decide what our response to them should be. For example, a particular class of error is the program which only copes with correctly formulated data. My own early experience of writing compilers is an example of this. These compilers did not behave well when presented with syntactically incorrect programs and the effect was to bring the machine down in all sorts of funny ways. Although correct programs were always run correctly, the compiler was not correct.

Such a problem arises again when many people cooperate on the programming of a piece of software. In fact the effect is compounded. Thus the study of errors, their causes and effects is important to the teaching of cooperative programming.

We shall study the problem, not from the point of view of error detection or of the programmer's response to errors but by a concern for the program's response to detected errors. We shall outline a coherent policy for designing software with this response in mind. The policy is more fully elaborated in [8]. Eventually such a policy should be implemented mechanically so that programs which do not obey it are rejected.

Handling errors often causes the distinction between modules in software to become blurred. Modules become too connected. For example, the information about the format of a particular magnetic tape is low-level. The information needed to access the files on

that tape is at a higher level. In general, programs which handle magnetic tapes do not separate these levels and run into problems for just this reason. It is important to make such distinctions in order to avoid errors.

Another important design point is to consider the probable evolution of the program. To begin with it will not be possible to predict the sort of errors which will cause trouble. Later, with experience, it will be possible to handle the most frequent type of error. This task itself will be easier and less error prone if the original design accepted its inevitability.

A final point is that it is important to identify the module which caused the error, so that the error can be referred to the appropriate programmer. Now if we use sophisticated module assembly techniques then this may be hard to do, since the distinction between modules in the final version may be considerably blurred.

3.1 A software "trap" feature

The way in which programs can be organized to exhibit the features described above is by the use of the software equivalent of a "trap". The "trap" allows the separation of three things:

i) code for the normal case
ii) code for error correction
iii) code for error detection

An area where this sort of organization has immediate application is the programming associated with handling input and output. If you write the normal case code and then add detection and correction code, in this situation you end up with a mess. The probability of error in this situation is quite high.

For example if we write a program for a virtual machine with a virtual tape unit, the normal case code (for the virtual machine) is written under the assumption that no error occurs. The responsibility for detecting errors in the actual tape unit appear in the elaboration of the virtual unit. The "trap" which such an error causes is signalled to the virtual machine by calling a routine provided at the level of the virtual machine. Thus the error correction (or response) resides in code separate from the normal case.

A module specification (of the virtual tape unit, for example) will have three parts:

1. Names of routines to be called if the preconditions for use of the unit are not met.
2. Names of routines to be called if the mechanism fails.
3. Description of the effect, if everything is satisfactory.

The first two parts may cause traps and where possible the design philosophy is that no changes will be made by the module before the trap is called. The importance of the organization established by the consistent use of traps is the ability of the modules to cope with the unusual.

Thus we place the responsibility for checking the misuse within the module and responsibility for doing something about it in the outside environment. A detected error is returned to the higher level (reflected) in terms which the caller knows about. It is no use, for example to tell the caller that you have just dropped a link from your list, if he does not (should not) even know you have got a list.

Errors are caused at one level and detected at another. When an error is detected it is either propagating downward, (in which case it is detected as an error of misuse), or propagating upward, in which case it will be detected as an error of mechanism. Upward propagating errors in fact have two sources. Either there has been a failure of lower level software of hardware (machinery) or a previously downward propagating error has been reflected.

An error of misuse is reflected by calling a trap routine which may assume responsibility for the error, and try and put things right, or may "pass the buck" by calling higher level trap routines. Early versions of a system may not recover in any real sense. However, one responsibility which a trap routing may not assume (this is a design point) is to abort the job. Because information important to recovery or diagnosis exists at higher levels it is necessary to allow abortion only at the highest level.

We have a classification into three types of error:

i) downward propagating - error of misuse
ii) upward propagating - error of mechanism
iii) upward propagating - reflected error of misuse

The next level of classification is a somewhat incomplete list of common error situations.

1. Violation of parameter value limitations:

   run time checks should only be omitted when it has been proved that it is impossible to violate them (e.g. because of checks at compile time).

2. Exceeding internal capacity limitations:

   Modules which provide a storage function will always have capacity limitations. In some situations it will be necessary to be a little over cautious, for example when the only convenient test is strictly "worst case".

3. Request for undefined information.

This list is obviously incomplete. In general we can summarize and design philosophy as the need for sufficiency in the error trap conditions in that they should guarantee that, if none of them applies, the module will perform according to its specification. Thus the conditions will in general be over cautious. By arranging that traps have a priority structure we can avoid calling many traps for the same error.

References

1. Parnas, D.L. "A Course on Software Engineering Techniques" March 1972.
2. Henderson, P. and Snowdon, R. "An Experiment in Structured Programming". BIT 12, 1 (1972) p. 38.
3. Hoare, C.A.R. "Proof of a Program: FIND". CACM Vol. 14, 1 (January, 1971) p. 39.
4. Parnas, D.L. "A Technique for Software Module Specification with Examples". CACM Vol. 15, 5 (May, 1972) p. 330.

5. Floyd, R.W. Treesort 3 Algorithm 245. CACM Vol. 7, 12 (December 1964) p. 701.

6. Parnas, D.L. "Information Distribution Aspects of Design Methodology" IFIP 1971 Ljublana Congress.

7. Parnas, D.L. "On the Criteria to be Used in Decomposing Systems into Modules" in CACM Vol. 15, (December, 1972).

8. Parnas, D.L. "Response to detected errors in well-structured programs" Carnegie Mellon T.R. July 1972.

9. Ashenhurst, R.P. (Editor) "Curriculum recommendations for graduate programs in Information Systems" CACM Vol. 15, 5 (May 1972) p. 363.

Discussion took place on each of three occasions after each of the sections.

Discussion after section 1.

Professor Ashenhurst complained that we did not really understand all aspects of how things are connected. Diagramming the human body as a stomach, liver, heart etc. lends itself to the sort of structuring which has just been described. However, if the nervous system or the skeletal system are diagrammed a different structuring is realized because both systems are in every part of the body. The nervous system is spread throughout the body, it is not localized and is thus much harder to represent by means of a block diagram. In this case, the performance is not in terms of what each element does but in terms of how the whole system works. The distinction between the two types of structuring is perhaps that what is being passed between modules in the first case is information and in the second case 'control' is passed between the elements of the system. The kind of structuring which has been discussed doesn't show this and we need a new way of seeing what control actions really are.

Professor Parnas replied that the problem with considering the human body as a typical system is that if one tried to maintain the body on the basis of textbook diagrams one would soon be in a lot of trouble.

Professor Dijkstra commented on Professor Ashenhurst's remark. He would draw his attention to the definition of a system as given by Anatol Holt, namely "A set of interrelated parts". In English this definition contains a carefully chosen ambiguity because 'part' can mean 'part of a whole' - a set of spatial relations or it can mean a 'part or role in a play' - a temporal relationship. Thus a system can be viewed within a set of time space axes. The interrelation between parts contains aspects of space and time.

Discussion after section 2

Professor Ashenhurst pointed out that one of the versions of Treesort regards the data structure both as a tree and as a straight sequence. He suggested that, while this type of dual structuring is useful, it admits that there is an underlying memory of bits, addresses, etc. This presents again the need to prevent people from falling into the temptation of using that fact to address the memory directly and make the whole thing machine dependent.

Professor Parnas replied that the specification of his tree module was based on the assumption that passage through the tree would only be by father-son sequences. It would therefore be very expensive to print all the leaves from left to right. However, if this facility was required, he could extend his specification, still hiding the implementation, to provide functions for accessing the leftmost, second leftmost, etc. leaves and the decision to provide this would be taken long before selecting the actual mode of implementation.

Professor Randell suggested that this approach is based on the implicit knowledge that an implementation of the specification would be possible and practical. He postulated that experience of what can be built influences the nature of the functions specified.

Professor Parnas considered that intuitive experience did influence his thinking to some extent. However, he tries very hard to look at the assumptions he makes and to get away from them as much as possible. He wants to achieve as high a degree of abstraction as possible such that in moving to a machine with a different technology, the specification need not be altered.

Dr. Scoins asked if Professor Parnas was not forced to give an example of a particular form in order to convey this to the students.

Professor Parnas replied that he is, at first. He shows them a stack and several different implementations and makes them write them. He makes them perform the experiment to show how interchangeable the alternatives can be. After a while, they learn to read the specifications and get used to them. They do not really ask to look at the implementation.

Discussion after section 3

Professor Verrijn-Stuart asked why the material presented here was relevant to the topic of conference. Professor Parnas replied that he was tempted to "trap" that one and pass it up a level to Brian Randell whose idea it was to invite him. However, he felt that people who write commercial systems ought to be exposed to such ideas because it is a point of view that is not seen in industry. He tried to choose examples which were from commercially oriented systems. Professor Ashenhurst said that this was just the sort of material that was covered by course C4 [9] but Professor Parnas said he could not tell that from reading the specification.

Professor Ashenhurst tackled the problem of completeness and asked if it was possible to handle each error locally when the organization described was looking for very definite errors and where, for example, numerical errors are very indefinite things. Professor Parnas agreed and observed that this was a very hard problem. He thought that this was a question of what goes into the section that detects errors of mechanism but said that this was the area we knew least about.

In relation to both questions Professor Colin observed that in business data processing you are bound to have blunders and he thought that this method of tackling errors was very important for the case when you have lots of data about.