Programming Techniques
R. Morris Editor

# On the Criteria To Be Used in Decomposing Systems into Modules

D.L. Parnas
Carnegie-Mellon University

**This paper discusses modularization as a mechanism for improving the flexibility and comprehensibility of a system while allowing the shortening of its development time. The effectiveness of a "modularization" is dependent upon the criteria used in dividing the system into modules. A system design problem is presented and both a conventional and unconventional decomposition are described. It is shown that the unconventional decompositions have distinct advantages for the goals outlined. The criteria used in arriving at the decompositions are discussed. The unconventional decomposition, if implemented with the conventional assumption that a module consists of one or more subroutines, will be less efficient in most cases. An alternative approach to implementation which does not have this effect is sketched.**

**Key Words and Phrases: software, modules, modularity, software engineering, KWIC index, software design**

**CR Categories: 4.0**



Author's address: Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213.

## Introduction

A lucid statement of the philosophy of modular programming can be found in a 1970 textbook on the design of system programs by Gouthier and Pont [1, ¶10.23], which we quote below:[1]

> A well-defined segmentation of the project effort ensures system modularity. Each task forms a separate, distinct program module. At implementation time each module and its inputs and outputs are well-defined, there is no confusion in the intended interface with other system modules. At checkout time the integrity of the module is tested independently; there are few scheduling problems in synchronizing the completion of several tasks before checkout can begin. Finally, the system is maintained in modular fashion; system errors and deficiencies can be traced to specific system modules, thus limiting the scope of detailed error searching.

Usually nothing is said about the criteria to be used in dividing the system into modules. This paper will discuss that issue and, by means of examples, suggest some criteria which can be used in decomposing a system into modules.

## A Brief Status Report

The major advancement in the area of modular programming has been the development of coding techniques and assemblers which (1) allow one module to be written with little knowledge of the code in another module, and (2) allow modules to be reassembled and replaced without reassembly of the whole system. This facility is extremely valuable for the production of large pieces of code, but the systems most often used as examples of problem systems are highly-modularized programs and make use of the techniques mentioned above.

[1] Reprinted by permission of Prentice-Hall, Englewood Cliffs, N.J.

## Expected Benefits of Modular Programming

The benefits expected of modular programming are: (1) managerial—development time should be shortened because separate groups would work on each module with little need for communication: (2) product flexibility—it should be possible to make drastic changes to one module without a need to change others; (3) comprehensibility—it should be possible to study the system one module at a time. The whole system can therefore be better designed because it is better understood.

## What Is Modularization?

Below are several partial system descriptions called *modularizations*. In this context "module" is considered to be a responsibility assignment rather than a subprogram. The *modularizations* include the design decisions which must be made *before* the work on independent modules can begin. Quite different decisions are included for each alternative, but in all cases the intention is to describe all "system level" decisions (i.e. decisions which affect more than one module).

## Example System 1: A KWIC Index Production System

The following description of a KWIC index will suffice for this paper. The KWIC index system accepts an ordered set of lines, each line is an ordered set of words, and each word is an ordered set of characters. Any line may be "circularly shifted" by repeatedly removing the first word and appending it at the end of the line. The KWIC index system outputs a listing of all circular shifts of all lines in alphabetical order.

This is a small system. Except under extreme circumstances (huge data base, no supporting software), such a system could be produced by a good programmer within a week or two. Consequently, none of the difficulties motivating modular programming are important for this system. Because it is impractical to treat a large system thoroughly, we must go through the exercise of treating this problem as if it were a large project. We give one modularization which typifies current approaches, and another which has been used successfully in undergraduate class projects.

### Modularization 1

We see the following modules:

**Module 1: Input.** This module reads the data lines from the input medium and stores them in core for processing by the remaining modules. The characters are packed four to a word, and an otherwise unused character is used to indicate the end of a word. An index is kept to show the starting address of each line.

**Module 2: Circular Shift.** This module is called after the input module has completed its work. It prepares an index which gives the address of the first character of each circular shift, and the original index of the line in the array made up by module 1. It leaves its output in core with words in pairs (original line number, starting address).

**Module 3: Alphabetizing.** This module takes as input the arrays produced by modules 1 and 2. It produces an array in the same format as that produced by module 2. In this case, however, the circular shifts are listed in another order (alphabetically).

**Module 4: Output.** Using the arrays produced by module 3 and module 1, this module produces a nicely formatted output listing all of the circular shifts. In a sophisticated system the actual start of each line will be marked, pointers to further information may be inserted, and the start of the circular shift may actually not be the first word in the line, etc.

**Module 5: Master Control.** This module does little more than control the sequencing among the other four modules. It may also handle error messages, space allocation, etc.

It should be clear that the above does not constitute a definitive document. Much more information would have to be supplied before work could start. The defining documents would include a number of pictures showing core formats, pointer conventions, calling conventions, etc. All of the interfaces between the four modules must be specified before work could begin.

This is a modularization in the sense meant by all proponents of modular programming. The system is divided into a number of modules with well-defined interfaces; each one is small enough and simple enough to be thoroughly understood and well programmed. Experiments on a small scale indicate that this is approximately the decomposition which would be proposed by most programmers for the task specified.

### Modularization 2

We see the following modules:

**Module 1: Line Storage.** This module consists of a number of functions or subroutines which provide the means by which the user of the module may call on it. The function call $CHAR(r,w,c)$ will have as value an integer representing the $c$th character in the $r$th line, $w$th word. A call such as $SETCHAR(r,w,c,d)$ will cause the $c$th character in the $w$th word of the $r$th line to be the character represented by $d$ (i.e. $CHAR(r,w,c) = d$). $WORDS(r)$ returns as value the number of words in

line *r*. There are certain restrictions in the way that these routines may be called; if these restrictions are violated the routines "trap" to an error-handling subroutine which is to be provided by the users of the routine. Additional routines are available which reveal to the caller the number of words in any line, the number of lines currently stored, and the number of characters in any word. Functions *DELINE* and *DELWRD* are provided to delete portions of lines which have already been stored. A precise specification of a similar module has been given in [3] and [8] and we will not repeat it here.

**Module 2: INPUT.** This module reads the original lines from the input media and calls the line storage module to have them stored internally.

**Module 3: Circular Shifter.** The principal functions provided by this module are analogs of functions provided in module 1. The module creates the impression that we have created a line holder containing not all of the lines but all of the circular shifts of the lines. Thus the function call *CSCHAR(l,w,c)* provides the value representing the *c*th character in the *w*th word of the *l*th circular shift. It is specified that (1) if $i < j$ then the shifts of line $i$ precede the shifts of line $j$, and (2) for each line the first shift is the original line, the second shift is obtained by making a one-word rotation to the first shift, etc. A function *CSSETUP* is provided which must be called before the other functions have their specified values. For a more precise specification of such a module see [8].

**Module 4: Alphabetizer.** This module consists principally of two functions. One, *ALPH*, must be called before the other will have a defined value. The second, *ITH*, will serve as an index. *ITH(i)* will give the index of the circular shift which comes *i*th in the alphabetical ordering. Formal definitions of these functions are given [8].

**Module 5: Output.** This module will give the desired printing of set of lines or circular shifts.

**Module 6: Master Control.** Similar in function to the modularization above.

### Comparison of the Two Modularizations

**General.** Both schemes will work. The first is quite conventional; the second has been used successfully in a class project [7]. Both will reduce the programming to the relatively independent programming of a number of small, manageable, programs.

Note first that the two decompositions may share all data representations and access methods. Our discussion is about two different ways of cutting up what *may* be the same object. A system built according to decomposition 1 could conceivably be identical *after assembly* to one built according to decomposition 2. The differences between the two alternatives are in the way that they are divided into the work assignments, and the interfaces between modules. The algorithms used in both cases *might* be identical. The systems are substantially different even if identical in the runnable representation. This is possible because the runnable representation need only be used for running; other representations are used for changing, documenting, understanding, etc. The two systems will not be identical in those other representations.

**Changeability.** There are a number of design decisions which are questionable and likely to change under many circumstances. This is a partial list.

1. Input format.
2. The decision to have all lines stored in core. For large jobs it may prove inconvenient or impractical to keep all of the lines in core at any one time.
3. The decision to pack the characters four to a word. In cases where we are working with small amounts of data it may prove undesirable to pack the characters; time will be saved by a character per word layout. In other cases we may pack, but in different formats.
4. The decision to make an index for the circular shifts rather that actually store them as such. Again, for a small index or a large core, writing them out may be the preferable approach. Alternatively, we may choose to prepare nothing during *CSSETUP*. All computation could be done during the calls on the other functions such as *CSCHAR*.
5. The decision to alphabetize the list once, rather than either (a) search for each item when needed, or (b) partially alphabetize as is done in Hoare's FIND [2]. In a number of circumstances it would be advantageous to distribute the computation involved in alphabetization over the time required to produce the index.

By looking at these changes we can see the differences between the two modularizations. The first change is confined to one module in both decompositions. For the first decomposition the second change would result in changes in every module! The same is true of the third change. In the first decomposition the format of the line storage in core must be used by all of the programs. In the second decomposition the story is entirely different. Knowledge of the exact way that the lines are stored is entirely hidden from all but module 1. Any change in the manner of storage can be confined to that module!

In some versions of this system there was an additional module in the decomposition. A symbol table module (as specified in [3]) was used within the line storage module. This fact was completely invisible to the rest of the system.

The fourth change is confined to the circular shift module in the second decomposition, but in the first decomposition the alphabetizer and the output routines will also know of the change.

The fifth change will also prove difficult in the first decomposition. The output module will expect the index to have been completed before it began. The alphabetizer module in the second decomposition was

designed so that a user could not detect when the alphabetization was actually done. No other module need be changed.

**Independent Development.** In the first modularization the interfaces between the modules are the fairly complex formats and table organizations described above. These represent design decisions which cannot be taken lightly. The table structure and organization are essential to the efficiency of the various modules and must be designed carefully. The development of those formats will be a major part of the module development and that part must be a joint effort among the several development groups. In the second modularization the interfaces are more abstract; they consist primarily in the function names and the numbers and types of the parameters. These are relatively simple decisions and the independent development of modules should begin much earlier.

**Comprehensibility.** To understand the output module in the first modularization, it will be necessary to understand something of the alphabetizer, the circular shifter, and the input module. There will be aspects of the tables used by output which will only make sense because of the way that the other modules work. There will be constraints on the structure of the tables due to the algorithms used in the other modules. The system will only be comprehensible as a whole. It is my subjective judgment that this is not true in the second modularization.

## The Criteria

Many readers will now see what criteria were used in each decomposition. In the first decomposition the criterion used was to make each major step in the processing a module. One might say that to get the first decomposition one makes a flowchart. This is the most common approach to decomposition or modularization. It is an outgrowth of all programmer training which teaches us that we should begin with a rough flowchart and move from there to a detailed implementation. The flowchart was a useful abstraction for systems with on the order of 5,000–10,000 instructions, but as we move beyond that it does not appear to be sufficient; something additional is needed.

The second decomposition was made using "information hiding" [4] as a criterion. The modules no longer correspond to steps in the processing. The line storage module, for example, is used in almost every action by the system. Alphabetization may or may not correspond to a phase in the processing according to the method used. Similarly, circular shift might, in some circumstances, not make any table at all but calculate each character as demanded. Every module in the second decomposition is characterized by its knowledge of a design decision which it hides from all others. Its interface or definition was chosen to reveal as little as possible about its inner workings.

## Improvement in Circular Shift Module

To illustrate the impact of such a criterion let us take a closer look at the design of the circular shift module from the second decomposition. Hindsight now suggests that this definition reveals more information than necessary. While we carefully hid the method of storing or calculating the list of circular shifts, we specified an order to that list. Programs could be effectively written if we specified only (1) that the lines indicated in circular shift's current definition will all exist in the table, (2) that no one of them would be included twice, and (3) that an additional function existed which would allow us to identify the original line given the shift. By prescribing the order for the shifts we have given more information than necessary and so unnecessarily restricted the class of systems that we can build without changing the definitions. For example, we have not allowed for a system in which the circular shifts were produced in alphabetical order, *ALPH* is empty, and *ITH* simply returns its argument as a value. Our failure to do this in constructing the systems with the second decomposition must clearly be classified as a design error.

In addition to the general criteria that each module hides some design decision from the rest of the system, we can mention some specific examples of decompositions which seem advisable.

1. A *data structure*, its internal linkings, *accessing procedures and modifying procedures* are part of a single module. They are not shared by many modules as is conventionally done. This notion is perhaps just an elaboration of the assumptions behind the papers of Balzer [9] and Mealy [10]. Design with this in mind is clearly behind the design of BLISS [11].
2. *The sequence of instructions necessary to call a given routine and the routine itself are part of the same module.* This rule was not relevant in the Fortran systems used for experimentation but it becomes essential for systems constructed in an assembly language. There are no perfect general calling sequences for real machines and consequently they tend to vary as we continue our search for the ideal sequence. By assigning responsibility for generating the call to the person responsible for the routine we make such improvements easier and also make it more feasible to have several distinct sequences in the same software structure.
3. The *formats of control blocks* used in queues in operating systems and similar programs *must be hidden* within a "control block module." It is conventional to make such formats the interfaces between various modules. Because design evolution forces frequent changes on control block formats such a decision often proves extremely costly.
4. *Character codes, alphabetic orderings, and similar data should be hidden* in a module for greatest flexibility.
5. The sequence in which certain items will be processed should (as far as practical) be hidden within a single module. Various changes ranging from equip-

ment additions to unavailability of certain resources in an operating system make sequencing extremely variable.

### Efficiency and Implementation

If we are not careful the second decomposition will prove to be much less efficient than the first. If each of the functions is actually implemented as a procedure with an elaborate calling sequence there will be a great deal of such calling due to the repeated switching between modules. The first decomposition will not suffer from this problem because there is relatively infrequent transfer of control between modules.

To save the procedure call overhead, yet gain the advantages that we have seen above, we must implement these modules in an unusual way. In many cases the routines will be best inserted into the code by an assembler; in other cases, highly specialized and efficient transfers would be inserted. To successfully and efficiently make use of the second type of decomposition will require a tool by means of which programs may be written as if the functions were subroutines, but assembled by whatever implementation is appropriate. If such a technique is used, the separation between modules may not be clear in the final code. For that reason additional program modification features would also be useful. In other words, the several representations of the program (which were mentioned earlier) must be maintained in the machine together with a program performing mapping between them.

### A Decomposition Common to a Compiler and Interpretor for the Same Language

In an earlier attempt to apply these decomposition rules to a design project we constructed a translator for a Markov algorithm expressed in the notation described in [6]. Although it was not our intention to investigate the relation between compiling and interpretive translators of a langugage, we discovered that our decomposition was valid for a pure compiler and several varieties of interpretors for the language. Although there would be deep and substantial differences in the final running representations of each type of compiler, we found that the decisions implicit in the early decomposition held for all.

This would not have been true if we had divided responsibilities along the classical lines for either a compiler or interpretor (e.g. syntax recognizer, code generator, run time routines for a compiler). Instead the decomposition was based upon the hiding of various decisions as in the example above. Thus register representation, search algorithm, rule interpretation etc. were modules and these problems existed in both compiling and interpretive translators. Not only was the decomposition valid in all cases, but many of the routines could be used with only slight changes in any sort of translator.

This example provides additional support for the statement that the order in time in which processing is expected to take place should not be used in making the decomposition into modules. It further provides evidence that a careful job of decomposition can result in considerable carryover of work from one project to another.

A more detailed discussion of this example was contained in [8].

### Hierarchical Structure

We can find a program hierarchy in the sense illustrated by Dijkstra [5] in the system defined according to decomposition 2. If a symbol table exists, it functions without any of the other modules, hence it is on level 1. Line storage is on level 1 if no symbol table is used or it is on level 2 otherwise. Input and Circular Shifter require line storage for their functioning. Output and Alphabetizer will require Circular Shifter, but since Circular Shifter and line holder are in some sense compatible, it would be easy to build a parameterized version of those routines which could be used to alphabetize or print out either the original lines or the circular shifts. In the first usage they would not require Circular Shifter; in the second they would. In other words, our design has allowed us to have a single representation for programs which may run at either of two levels in the hierarchy.

In discussions of system structure it is easy to confuse the benefits of a good decomposition with those of a hierarchical structure. We have a hierarchical structure if a certain relation may be defined between the modules or programs and that relation is a partial ordering. The relation we are concerned with is "uses" or "depends upon." It is better to use a relation between programs since in many cases one module depends upon only part of another module (e.g. Circular Shifter depends only on the output parts of the line holder and not on the correct working of *SETWORD*). It is conceivable that we could obtain the benefits that we have been discussing without such a partial ordering, e.g. if all the modules were on the same level. The partial ordering gives us two additional benefits. First, parts of the system are benefited (simplified) because they use the services of lower[2] levels. Second, we are able to cut off the upper levels and still have a usable and useful product. For example, the symbol table can be used in other applications; the line holder could be the basis of a question answering system. The existence of the hierarchical structure assures us that we can "prune" off the upper levels of the tree and start a new tree on the old trunk. If we had designed a system in which the "low level" modules made some use of the "high level" modules, we would not have the hierarchy, we would find it much harder to remove portions of the system, and "level" would not have much meaning in the system.

[2] Here "lower" means "lower numbered."

Since it is conceivable that we could have a system with the type of decomposition shown in version 1 (important design decisions in the interfaces) but retaining a hierarchical structure, we must conclude that hierarchical structure and "clean" decomposition are two desirable but *independent* properties of a system structure.

## Conclusion

We have tried to demonstrate by these examples that it is almost always incorrect to begin the decomposition of a system into modules on the basis of a flowchart. We propose instead that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others. Since, in most cases, design decisions transcend time of execution, modules will not correspond to steps in the processing. To achieve an efficient implementation we must abandon the assumption that a module is one or more subroutines, and instead allow subroutines and programs to be assembled collections of code from various modules.

Received August 1971; revised November 1971

### References

1. Gauthier, Richard, and Pont, Stephen. *Designing Systems Programs*, (C), Prentice-Hall, Englewood Cliffs, N.J., 1970.
2. Hoare, C. A. R. Proof of a program, FIND. *Comm. ACM 14*, 1 (Jan. 1971), 39–45.
3. Parnas, D. L. A technique for software module specification with examples. *Comm. ACM 15*, 5 (May, 1972), 330–336.
4. Parnas, D. L. Information distribution aspects of design methodology. Tech. Rept., Depart. Computer Science, Carnegie-Mellon U., Pittsburgh, Pa., 1971. Also presented at the IFIP Congress 1971, Ljubljana, Yugoslavia.
5. Dijkstra, E. W. The structure of "THE"-multiprogramming system. *Comm. ACM 11*, 5 (May 1968), 341–346.
6. Galler, B., and Perlis, A. J. *A View of Programming Languages*, Addison-Wesley, Reading, Mass., 1970.
7. Parnas, D. L. A course on software engineering. Proc. SIGCSE Technical Symposium, Mar. 1972.
8. Parnas, D. L. On the criteria to be used in decomposing systems into modules. Tech. Rept., Depart. Computer Science, Carnegie-Mellon U., Pittsburgh, Pa., 1971.
9. Balzer, R. M. Dataless programming. Proc. AFIPS 1967 FJCC, Vol. 31, AFIPS Press, Montvale, N.J., pp. 535–544.
10. Mealy, G. H. Another look at data. Proc. AFIPS 1967 FJCC, Vol. 31, AFIPS Press, Montvale, N.J., pp. 525–534.
11. Wulf, W. A., Russell, D. B., and Habermann, A. N. BLISS, A language for systems programming. *Comm. ACM 14*, 12 (Dec. 1971), 780–790.